

Automated Localisation of a Mixed Boolean Arithmetic Obfuscation Window in a Program Binary

Antoine De Schrijver

Student number: 01506917

Supervisors: Prof. dr. ir. Bjorn De Sutter, Dr. Bart Coppens

Counsellor: Dr. ir. Bert Abrath

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2020-2021

Automated Localisation of a Mixed Boolean Arithmetic Obfuscation Window in a Program Binary

Antoine De Schrijver

Student number: 01506917

Supervisors: Prof. dr. ir. Bjorn De Sutter, Dr. Bart Coppens

Counsellor: Dr. ir. Bert Abrath

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2020-2021

Preface

This dissertation marks my final step as a student at Ghent University. It combines my interest in cybersecurity with my acquired knowledge in artificial intelligence. Despite this unusual year, I am very grateful I got to work with such enthusiastic supervisors. The pleasant online meetings were a delightful change in the stay-at-home invariance.

I want to express my sincere gratitude towards the following people. First, I'd like to thank Prof. dr. ir. Bjorn De Sutter for introducing me to the field of reverse engineering with his enlightening course on software hacking and protection. I would also like to thank him, Dr. Bart Coppens and Dr. ir. Bert Abrath for their insights and support throughout this work.

I would also like to thank my family and my girlfriend for supporting me throughout my studies. I am sure that without their support and encouragement, I would never have made it this far. Lastly I want to thank my friends, old and new, for the great times we had during our university years.

The author(s) gives (give) permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In all cases of other use, the copyright terms have to be respected, in particular with regard to the obligation to state explicitly the source when quoting results from this master dissertation.

Ghent, August 25th 2021

Automated Localisation of a Mixed Boolean-Arithmetic Obfuscation Window in a Program Binary

Antoine De Schrijver

Supervisors: Prof. dr. ir. Bjorn De Sutter Dr. Bart Coppens

Counsellor: Dr. ir. Bert Abrath

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2020-2021

Abstract Mixed Boolean-Arithmetic (MBA) is a mechanism for data obfuscation. It transforms constants and simple expressions into expressions that contain a mix of boolean and arithmetic operations. Due to the lack of general rules when mixing these two types, MBA-obfuscations manage to thwart most static and dynamic program analysis techniques. Recently, program synthesis proved to be quite successful in deobfuscating MBA transformations. By analysing the input-output behaviour of an MBA expression, a synthesiser can reconstruct a simplified version with the same semantics, essentially bypassing the obfuscation. However, the different program synthesis approaches do not consider the issue of locating the MBA-obfuscation within the program. Therefore, this thesis proposes a method to locate MBA obfuscations within a program binary. To achieve this, we first derive all possible segments in a program binary. We then leverage the power of supervised classification to flag the MBA expressions. We train this classifier by constructing a dataset of MBA-obfuscated functions and constructing a featureset inspired by the natural language processing field. Our results demonstrate that our approach manages to locate 89% of the MBA-obfuscated segments. This work is a first step in the localisation of MBA obfuscations. The proposed classifier and featuresets can also serve as a basis for the localisation and classification of other obfuscations.

Keywords software protection, obfuscation, mixed boolean-arithmetic, classification, localisation

Automated Localisation of a Mixed Boolean-Arithmetic Obfuscation Window in a Program Binary

Antoine De Schrijver

Supervisors: Prof. dr. ir. Bjorn De Sutter, Dr. Bart Coppens
Counsellor: Dr. ir. Bert Abrath

Abstract

Mixed Boolean-Arithmetic (MBA) is a mechanism for data obfuscation. It transforms constants and simple expressions into expressions that contain a mix of boolean and arithmetic operations. Due to the lack of general rules when mixing these two types, MBA-obfuscations manage to thwart most static and dynamic program analysis techniques. Recently, program synthesis proved to be quite successful in deobfuscating MBA transformations. By analysing the input-output behaviour of an MBA expression, a synthesiser can reconstruct a simplified version with the same semantics, essentially bypassing the obfuscation. However, the different program synthesis approaches do not consider the issue of locating the MBA-obfuscation within the program. Therefore, this thesis proposes a method to locate MBA obfuscations within a program binary. To achieve this, we first derive all possible segments in a program binary. We then leverage the power of supervised classification to flag the MBA expressions. We train this classifier by constructing a dataset of MBA-obfuscated functions and constructing a featureset inspired by the natural language processing field. Our results demonstrate that our approach manages to locate 89% of the MBA-obfuscated segments. This work is a first step in the localisation of MBA obfuscations. The proposed classifier and featuresets can also serve as a basis for the localisation and classification of other obfuscations.

Keywords: software protection, obfuscation, mixed boolean-arithmetic, classification, localisation

1. Introduction

Code obfuscation is the process of transforming a program into a more complex equivalent. While the semantics stay the same, the program's complexity increases significantly, hampering the analysis of the code [1]. This technique is widely used in software protection against reverse engineering attacks. It is an essential protection mechanism in scenario's where the attacker has complete control over the execution environment of the software, called a Man-At-The-End (MATE) Attack [2]. MATE attacks can either be used for malicious purposes, like the removal of Digital Rights Management (DRM) [3, 4], or benign goals, like malware inspection [3]. Hence, there is continuous research into obfuscation mechanisms as well as deobfuscation tools that break these mechanisms.

Over the years, researchers have devised various obfuscation

mechanisms to counter MATE attacks. They can roughly be divided into three categories: layout obfuscation, control flow obfuscation, and data obfuscation [5]. Layout obfuscation removes relevant information, like object names, without altering the behaviour of the code. Control flow obfuscations aim to change the program's flow by, e.g. inserting opaque predicates or flattening the control flow. Data obfuscation tries to hide data within the program, like important outcomes or secret keys.

This thesis will focus on a mechanism for data obfuscation called Mixed Boolean-Arithmetic (MBA). This technique hides simple expressions and constants by transforming them into expressions that contain a mix of boolean (e.g. *and*, *or*) and arithmetic (e.g. *+*, ***) operators [6, 7]. This mix proves difficult to simplify to the original expression.

In recent literature, program synthesis is put forward as a tool to undo certain obfuscations, including MBA. Blazytko et al. developed a black box synthesis framework called SYNTIA that automatically reconstructs obfuscated segments based on their input-output behaviour [8]. David et al. followed up on this research with QSYNTH that also take context into account to improve the synthesis performance [9]. While these tools prove effective in the deobfuscation of MBA expressions, they do not consider the problem of locating these segments within a program binary.

To improve the automation of MBA deobfuscation, we develop a method to locate MBA segments within a program binary based on classification using supervised machine learning. We derive all the possible segments of a program binary and classify them as MBA or not. Tools, like SYNTIA or QSYNTH, can then synthesise the MBA segments to obtain the original expression. With this purpose in mind, we make the following contributions in this thesis:

1. We devise a method to obtain all the possible segments from a program binary.
2. We gather a dataset consisting of 1400 MBA-obfuscated functions and 3075 unobfuscated functions.
3. We design a method for constructing *register-dependency N-grams* from a program binary.
4. We build a classifier that is able to differentiate between an unobfuscated segment and an MBA expression. We then locate MBA expressions by classifying all the possible segments in a program binary.

2. Background

Before getting into the design of our work, we provide some technical background on the topics relevant to this thesis.

2.1. Mixed Boolean-Arithmetic

In general, any expression consisting of both arithmetic operators (+, *, ...) and boolean operators (and, xor, ...) can be referred to as a mixed boolean-arithmetic expression. The use of mixed boolean-arithmetic for obfuscation was formalized by Zhou et al. [6]. They defined *polynomial MBA expressions* as follows:

Definition 1 (Polynomial MBA [6][7]) *An expression E of the form*

$$E = \sum_{i \in I} a_i \left(\prod_{j \in J_i} e_{i,j}(x_0, \dots, x_{t-1}) \right)$$

where the arithmetic sum and product are modulo 2^n , a_i are constants in $\mathbb{Z}/2^n\mathbb{Z}$, $e_{i,j}$ are bitwise expressions of variables x_0, \dots, x_{t-1} in $\{0, 1\}^n$, $I \subset \mathbb{Z}$ and for all $i \in I$, $J_i \subset \mathbb{Z}$ are finite index sets, is a polynomial Mixed Boolean-Arithmetic (MBA) expression. A linear MBA expression is a polynomial MBA expression of the form

$$\sum_{i \in I} a_i e_i(x_0, \dots, x_{t-1})$$

Using this definition, Zhou et al. formalised a way to rewrite simple expressions and constants as MBA expressions, effectively obfuscating the original expression.

The strength of the MBA transformation lies in the incompatibility between the arithmetic and bit-wise operators. No general rules (like distributivity or associativity) exist to simplify these expressions. As a result, it is resilient against most static and dynamic analysis. MBA is used by various commercial and academic obfuscators, like Quarkslab[10], Irdeto [11] and Tigress [12].

2.2. Program synthesis

Program synthesis is the task of automatically constructing a program based on a given high-level specification [13]. While it has many different applications, it has recently been proposed as a way to deobfuscate programs. For deobfuscation, program synthesis can be used to reconstruct an obfuscated code segment based solely on specific observations, like input-output behaviour. A synthesiser tries to produce a program that retains the specified properties of the obfuscated code yet is more readable for a reverse engineer. The result is a code segment that is easier to understand than the obfuscated program, essentially bypassing the effects of the obfuscation transformation.

Blazytko et al. proposed such a solution, called SYNTIA[8]. This tool reconstructs fragments of programs based on the input-output behaviour of that segment. It essentially considers the segment as a black box. SYNTIA first generates random inputs and looks at the corresponding outputs. These

I/O pairs are then used to retrieve the semantics of the segment by using a heuristic based on MCTS trees. David et al. improved on this approach with QSYNTH[9]. They combine the black box approach of SYNTIA with Dynamic Symbolic Execution to simplify the problem further. Both SYNTIA and QSYNTH prove effective at undoing MBA-obfuscations, virtualization obfuscations and ROP chains. However, they need the exact start and end of the obfuscated segment to generate correct input-output pairs. They do not consider the problem of locating said obfuscations in the program.

2.3. Supervised Classification

A supervised machine learning model tries to approximate the unknown function $y = f(\mathbf{x})$ based on a training set of example input-output pairs $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)$ [14]. The vector of inputs $x_1, x_2, \dots, x_k \in \mathbf{x}_i$ are called *features*, the outputs y_i are referred to as *labels*. In a classification scenario, the labels are discrete. For unknown data, a model predicts a label based on the features of that data. These features are derived from the data through a process known as *feature extraction*.

3. Design

To locate the MBA segments in a program binary, we propose the following design. First, we derive all the possible code windows from a program binary. A supervised machine learning model then automatically classifies each of these windows as MBA-obfuscated or unobfuscated. After this step, we now know the location of the MBA-obfuscated segments. These segments can then be fed to a program synthesiser, like SYNTIA to obtain the original expression.

4. Retrieving all possible segments

To obtain all the possible segments in a program binary, we apply a *varying sliding window* algorithm. Every iteration, we obtain a different code segment of the program binary. The first window starts at the first instruction of the binary and is `minimum_size` instructions long. The next window is one instruction longer than that. This incremental growth continues until the window has a size of `maximum_size` instructions. At this point, the window moves an instruction and starts at the second instruction, again of size `minimum_size` instructions. This window grows again until the maximum size is reached and moves again. This algorithm continues until the end of the code segment is reached. An example is illustrated in Figure 1.

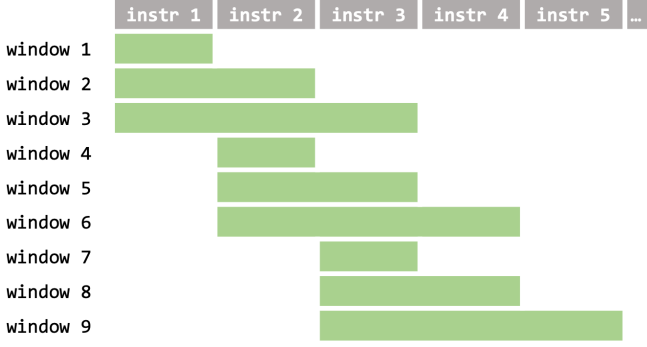


Figure 1. Example of the *varying sliding window* algorithm for a `minimum_size` of 1 and a `maximum_size` of 3. This algorithm continues to create new windows until the end of the binary is reached.

5. Feature Extraction

The raw program binaries need to be turned into useful features for the classifier. This process is called feature extraction. It is essential that these features are informative for the task at hand. For this problem, the chosen features need to contain the necessary information to indicate whether a specific segment of a program binary is an MBA expression or not. To achieve this, we first reconstruct the assembly of the program binary using *objdump* [15]. We then devise two types of features, inspired by the field of natural language processing (NLP).

5.1. Bag-of-instructions

A traditional NLP *bag-of-words* approach sees a text document as an unordered set of words with their position ignored [16]. Only the frequency of these words is kept. In the context of a program binary, we can use the assembly instructions instead of words. We will refer to this technique as a *bag-of-instructions*. For a given segment, every occurrence of each instruction is counted. The frequency of each instruction is then calculated and used as a feature for the machine learning model.

A downside of the *bag-of-instructions* model is the fact that it takes no context into account. Segments with the same instructions but with a different order receive the same label. Therefore, we turn to *N-grams* instead.

5.2. Register Dependency N-grams

In natural language processing, *N-grams* try to encapsulate the order of words into machine learning features. An *N-gram* is a combination of N consecutive words. For text classification, the frequency of each *N-gram* in a text is registered and then used as a feature. Therefore, an *N-gram* can be seen as a generalisation of the *bag-of-words* scenario, with a *bag-of-words* corresponding to a *1-gram* or *unigram*. Intuitively, information about the order of words is important for understanding a text segment: words that follow each other are often related in a sentence. In assembly code, this is not necessarily the case. Consecutive instructions can act on different registers and memory addresses, being placed in a particular order because of compiler optimisations.

Instead of creating *N-grams* based solely on the instruction order, we also take register dependencies into account. Instructions that form a read-after-write dependency chain can form an *N-gram*. As mixed boolean-arithmetic expressions mix boolean and arithmetic instructions, their *N-grams* should be distinguishable from ordinary *N-grams*. Like in natural language processing research, we will limit ourselves to 2-grams and 3-grams (often called bigrams and trigrams). Larger *N-grams* result in a bigger and sparser featureset, both attributes that will lead to overfitting and worse performing models.

5.2.1. DERIVING *N-grams*

To create these *N-grams* from the program binary, we use a custom *pintool* with *PIN* [17]. The *pintool* dynamically executes the program to derive the read-after-write register dependencies of the program. Therefore, it is essential that the segment containing the MBA expression is actually executed.

To efficiently retrieve all *N-grams* of a specific segment, we model all the register dependencies of a program binary as a directed graph. Nodes correspond to instruction addresses, and edges indicate the read-after-write register dependencies between them. For a specific segment, we can take the subgraph that contains the instructions of that segment. This is illustrated in Figure 2. An *N-gram* then corresponds to a walk of length N . With this method, we can obtain all the *N-grams* for a given segment.

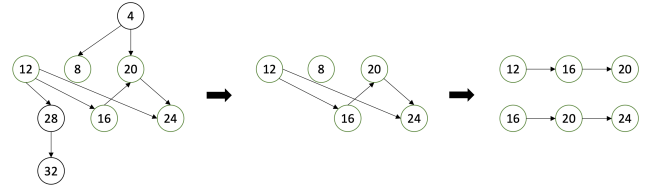


Figure 2. Illustration of how *N-grams* of a specific segment, ranging from instruction 8 to 24, are devised (here for $N=3$). First, all the register dependencies of the program binary are mapped to a graph. Then the subgraph is taken that only contains the instructions of the segment. Lastly, all the paths of length N are considered.

Instead of modelling raw register dependencies, we alter our *pintool* to model *shortcut dependencies*. When an instruction uses a value that is copied by a `mov`, it is dependent on this `mov` instruction. While this is technically correct, we are interested in the dependency between the instruction that last wrote to the `mov` source and the instruction after the `mov`. *Shortcut dependencies* do not consider the intermediary `mov` instructions and show the dependency between the prior instruction and the latter instruction instead.

6. Data

We need representative training data to get a working classifier. For this, we first explore existing data sources and conclude with the need for a new dataset.

6.1. Datasets

To train the classifier, we use three data sources. The first dataset comes from the SYNTIA framework [18]. It consists of 500 small functions and 500 obfuscated variants, obtained by using the Tigress *encode arithmetic* transform. The second dataset comes from the QSYNTH dataset [19]. It contains 500 small functions and their obfuscated counterparts, similar to the SYNTIA dataset. These two datasets were conceived to evaluate the performance of the program synthesis frameworks. While they are perfect for this purpose, they do not reflect real-life program binaries. Therefore, we create a new dataset based on a public Github repository that is called THE ALGORITHMS (referred to as ALGO for brevity). This repository is comprised of 300 small C programs that implement various computer science algorithms, like number conversions, hash calculations, and data structures. These small programs are turned into a dataset by obfuscating the functions within them with Tigress *encode arithmetic* transform and compiling both the original and the obfuscated versions using *GCC*. After removing erroneous program binaries and failed obfuscations, we end up with the ALGO dataset consisting of 2075 unobfuscated functions and 400 MBA-obfuscated functions.

6.2. Funcions vs. Expressions

While our dataset consists of original and MBA-obfuscated functions, we want to be able to classify MBA-obfuscated expressions. These segments do not necessarily correspond to an entire function. However, labelling every MBA-obfuscated expression would require manual analysis of every program binary. Instead, we opt to label entire functions as original or MBA-obfuscated. This can be done automatically with the debug information in the program binaries.

For the SYNTIA and QSYNTH dataset, every function computes one expression. When obfuscated, this function is roughly similar to one obfuscated MBA expression. This is not the case for the ALGO dataset, where functions contain logic besides expressions. One function will thus likely contain multiple MBA-obfuscated expressions, together with some unobfuscated logic (e.g. for output).

6.3. Static vs. Dynamic dataset

In section 5.2.1 we explained the use of dynamic analysis to retrieve the register dependency N-grams. We stressed that the MBA expression must be executed to obtain the N-grams for that segment. This means that we need to define inputs for all the program binaries in each dataset. Otherwise, these programs will not execute anything meaningful and return an error, resulting in no N-grams.

The inputs for the SYNTIA and QSYNTH datasets are straightforward: all the functions require five input numbers to return a computed result. However, for the ALGO dataset, every algorithm expects different inputs. This requires an extensive manual process for all the available programs. Due to time constraints, we defined the inputs for 30 programs. This cuts down the usable ALGO dataset to 96 obfuscated functions and 599 unobfuscated functions.

6.4. Train-Test Split

To train the classifier, the data must first be split into a train set and a test set. Every dataset is divided according to an 80%-20% train-test ratio. To avoid data-leakage, duplicates were removed and every obfuscated function is in the same set as its unobfuscated counterpart. In summary, we end up with the data distribution shown in Table 1.

Dataset	Set	#MBA	#Original
SYNTIA + QSYNTH	train	1170	874
	test	113	92
ALGO dynamic	train	140	1134
	test	13	112
ALGO static	train	294	1607
	test	73	387

Table 1. Overview of the class distribution for the different datasets after removing duplicates and keeping original and obfuscated functions in the same set. The values represent the amount of functions

7. Classifier

7.1. Multinomial Naive Bayes Classifier

We build our classifier using a multinomial naive Bayes classifier. This model is often used in natural language processing with the *bag-of-words* or *N-gram* features. The classifier works by choosing the class with the maximum probability given the input features:

$$y = \operatorname{argmax}_y P(y|x_1, x_2, \dots, x_n) \quad (1)$$

Due to the naive Bayes assumption that the features are not dependent on each other given a certain class, this problem is simplified to:

$$y = \operatorname{argmax}_y P(y) \prod_{i=1}^n P(x_i | y) \quad (2)$$

The prior $P(y)$ and the posteriors $P(x_i|y)$ can now be inferred from the training data.

The features for this model are the raw counts of the different tokens. To every token, a constant count is also added to prevent multiplications with posteriors of value 0. This process is called *Laplace Smoothing* [20].

The naive Bayes model is based on the naive assumption that features are not dependent on each other, given a certain class. While this assumption does not hold for our data, this model has empirically been shown to work even when the assumption is not valid [**naiveBayes**’ study].

7.2. Evaluation metric

Due to the imbalances in the data, we opt to use *Matthews Correlation Coefficient* (MCC) [21] as a metric instead instead of *accuracy*. Accuracy alone can mislead the observer when class imbalances are present, as overpredicting the bigger class can still yield good accuracy. The MCC metric is more robust in this scenario. The MCC formula is shown below. MCC takes all the elements of the confusion matrix into account,

compared to only the true positives and the true negatives in the case of accuracy. It yields a value between -1 and 1, where a higher score is indicative of a better model.

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

7.3. Results

The performance of the classifier is shown in Table 2. Both the *bag-of-instructions* and the *bigrams* perform quite well for the separate datasets. The bad performance when working with trigrams is likely caused by the sparseness of the featureset. One obfuscated function often only contains a few trigrams, especially in the case of the SYNTIA and QSYNTH dataset. The ALGO dataset does not suffer as hard because the functions are bigger and contain more trigrams. When combining these two datasets, the small class probabilities of the SYNTIA+QSYNTH dataset are in stark contrast with the bigger class probabilities of the ALGO dataset, resulting in even worse performance when combined. However, combining the two sets should result in a model that generalises better on unseen data.

Dataset	Instructions	Bigrams	Trigrams
Syntia + Qsynth	0.65	0.70	0.51
Algo	0.96	0.90	0.87
All	0.69	0.49	0.42

Table 2. MCC scores for the multinomial naive Bayes classifier using the token count as features. The rows indicate the dataset on which the model was trained and tested. The columns show the used featuresets.

8. Evaluation

With the trained classifier, we now want to assess its performance for the problem at hand: locating MBA expressions in a program binary by classifying every possible segment. To evaluate the performance, we devise a dataset consisting of 30 program binaries that represent functions similar to the three datasets.

For every program binary, we manually label the segments that correspond to an MBA-obfuscated expression with the help of debug information from the compilation. Contrarily to the training datasets, these segments do not necessarily correspond to entire functions. One function can contain multiple MBA-obfuscated expressions. Those need to be separately located in order to be simplified by program synthesis.

To test the classifier’s performance, we generate all the possible segments of the evaluation program binaries using the *varying sliding window* process with a `minimum_size` of 5 and a `maximum_size` of 100. We categorise every segment depending on what kind of instructions they contain:

1. Segments that correspond exactly to an MBA expression.
2. Segments that only contain part of an MBA expression.
3. Segments that are bigger than, but completely contain an MBA expression.
4. Segments that contain part of an MBA expression, along with some unobfuscated code.
5. Segments that contain no MBA expressions.

The different categories are illustrated in Figure 3.

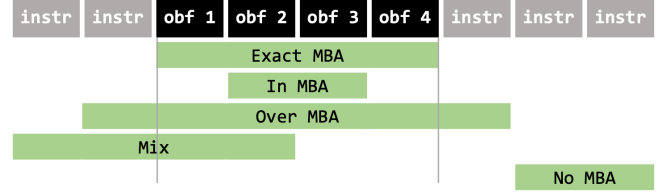


Figure 3. Visualisation of the different type of segments. The MBA-obfuscated segment starts at obf1 and ends at obf4. The green blocks show one segment.

We now label every segment using the naive Bayes classifier and look at the results for every category. We do this for the *bag-of-instructions*, the *bigrams* and the *trigrams*. The results are shown in Table 3.

The values should be interpreted as follows: for the *exact MBA* category, we want the percentage to be as high as possible in order to label as many MBA expressions as possible correctly. The *no MBA* values should be as low as possible: these segments offer nothing interesting when synthesised. The *mix* values are preferably also low; while they can indicate to the reverse engineer that an MBA segment is nearby, the synthesis result will not be meaningful. The same reasoning holds for the *In MBA* category. However, this category will be highly correlated with the *Exact MBA* category. The results of the *Over MBA* category are hardest to understand. Some of these segments will still produce a meaningful synthesis result, but most of them will not reveal the semantics of the obfuscated segments.

The model seems to perform best with the *bag-of-instructions* features. This is likely due to the difference in the amount of data: it only needs static program binaries for this featureset, compared to dynamic execution for the bigrams and trigrams. Hence, the entire ALGO dataset can be used. Nevertheless, the *bigram* and *trigram* model still deliver good results.

All three feature types are able to distinguish between unobfuscated segments and segments containing MBA obfuscations. Notably, 95% to 99% of the non obfuscated segments are classified correctly. Of the exact segments, 80% to 89% is classified correctly. For the other categories (in MBA, over MBA, mix), the results are mixed. While they are preferably classified as unobfuscated code, they do reveal to the reverse-engineer that there is an MBA expression nearby.

9. Conclusion

In this thesis, we proposed a method to locate mixed boolean-arithmetic obfuscations in a program binary. We achieved this by constructing a Multinomial Naive Bayes classifier and classifying each possible segment of a program binary as either unobfuscated, or MBA-obfuscated. For the classifier features, we explored the *bag-of-instructions*, the *register dependency bigrams* and the *register dependency trigrams*. We trained these models on a dataset comprising of the SYNTIA dataset, the QSYNTH dataset and a self-constructed ALGO dataset.

The *bag-of-instructions* resulted in the best performance when tested on an evaluation set of 30 program binaries. However, this is most likely due to the difference in the amount of training data compared to the *bigram* and *trigram* model.

Our best model correctly labels 89% of the MBA-obfuscations and 99% of the segments that contain no obfuscations. For the segments that are partly obfuscated, partly unobfuscated, the results vary. However, these segments do indicate that an MBA expression is nearby.

Featureset	Exact MBA	In MBA	Over MBA	Mix	No MBA	Total
Segment amount	41	1632	38523	24063	256943	321202
Bag-of-instructions	89 %	41 %	36 %	26 %	1 %	9 %
Bigrams	80 %	89 %	94 %	66 %	5 %	21 %
Trigrams	80 %	66 %	93 %	56 %	2 %	17 %

Table 3. Performance of the classifier on the evaluation set. The first row shows the amount of segments for each category. The other rows show the percentage of segments that were classified as MBA-obfuscated for each category.

10. Future Work

Our approach is based on supervised machine learning classification. In the scope of this thesis, a classifier was used to locate MBA-obfuscated segments. However, with the procured data and constructed features, this classifier can be extended to the classification and localisation of other types of obfuscation. The featureset can also be expanded by taking a look at the memory dependencies between instructions.

For this particular research, a next step could be the omission of the *sliding window* component. Instead, the following segmentation approach can be researched: train a classifier to classify an MBA-obfuscated expression’s first and last instruction. In this way, the MBA-obfuscated segment can be located in the program binary. The bigrams and trigrams can serve as features for each instruction. These dependency chains give a sense of previous and future context to each instruction.

In general, machine learning processes requires meaningful data in order to be trained and evaluated correctly. This is a bottleneck for the adaptation of machine learning in the field of reverse engineering. This field could benefit from more data sources to drive this kind of research forward.

References

- [1] Christian Collberg et al. *A taxonomy of obfuscating transformations*. Tech. rep. Citeseer, 1997.
- [2] Paolo Falcarin et al. “Guest editors’ introduction: Software protection”. In: *IEEE Software* 28.2 (2011).
- [3] Jasvir Nagra et al. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.
- [4] Camille Mougey et al. “DRM obfuscation versus auxiliary attacks”. In: *Recon conference*. 2014.
- [5] Maurizio Leotta et al. “A Family of Experiments to Assess the Impact of Page Object Pattern in Web Test Suite Development”. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2020.
- [6] Yongxin Zhou et al. “Information hiding in software with mixed boolean-arithmetic transforms”. In: *International Workshop on Information Security Applications*. Springer. 2007.
- [7] Ninon Eyrolles. “Obfuscation with Mixed Boolean-Arithmetic Expressions: reconstruction, analysis and simplification tools”. PhD thesis. Université Paris-Saclay, 2017.
- [8] Tim Blazytko et al. “Syntia: Synthesizing the semantics of obfuscated code”. In: *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 2017.
- [9] Robin David et al. “QSynth-A Program Synthesis based Approach for Binary Code Deobfuscation”. In: *BAR 2020 Workshop*. 2020.
- [10] Quarkslab. *Quarks App Protect*. <https://quarkslab.com/quarks-appshield-app-protect/>.
- [11] Irdeto. *Cloakware By Irdeto*. <https://irdeto.com/cloakware-by-irdeto/>.
- [12] University of Arizona. *The Tigress C Obfuscator*. <http://tigress.wtf>.
- [13] Sumit Gulwani. “Dimensions in program synthesis”. In: *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. 2010.
- [14] S. Russell et al. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2016.
- [15] GNU Project. *Obfdump - display information from object files*. <https://www.gnu.org/software/binutils/>.
- [16] Daniel Jurafsky et al. *Speech and Language Processing - An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition (3rd ed. draft)*. <https://web.stanford.edu/~jurafsky/slp3>. 2020.
- [17] Intel. *Pin - A Dynamic Binary Instrumentation Tool*. <https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html>.
- [18] Tim Blazytko et al. *Syntia Framework Github Repository*. <https://github.com/RUB-SysSec/syntia>.
- [19] Robin David et al. *Qsynth Dataset Github Repository*. <https://github.com/werew/qsynth-artifacts>.
- [20] Daniel Jurafsky et al. *Speech and Language Processing (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2009.
- [21] Brian W Matthews. “Comparison of the predicted and observed secondary structure of T4 phage lysozyme”. In: *Biochimica et Biophysica Acta (BBA)-Protein Structure* 405.2 (1975).

Contents

List of Figures	xxi
List of Tables	xxiii
List of Acronyms	xxv
1 Introduction	1
1.1 Reverse Engineering	1
1.2 Man-At-The-End Attack	2
1.3 Thesis Outline	2
2 Background	3
2.1 Software Protection	3
2.1.1 Motivation	4
2.1.2 Categories	4
2.2 Code Obfuscation	4
2.2.1 Metrics for obfuscation	5
2.3 Program Analysis	5
2.3.1 Graphical representations	6
2.3.2 Static analysis	8

2.3.3	Dynamic analysis	9
2.4	Symbolic Execution	10
2.5	Obfuscation types	11
2.5.1	Opaque Predicates	11
2.5.2	Function inlining/outlining	12
2.5.3	Control flow flattening	12
2.5.4	Mixed Boolean-Arithmetic	13
2.5.5	Virtualisation obfuscation	14
2.6	Program Synthesis	14
2.6.1	Expressing the high-level specification	15
2.6.2	Search Space	15
2.6.3	Search technique	15
2.6.4	Concrete implementations	17
3	Thesis Objective	21
3.1	Thesis Goal	21
3.2	Thesis Scope	21
3.3	Thesis Outline	22
4	Mixed Boolean-Arithmetic	23
4.1	Polynomial Mixed Boolean-Arithmetic	23
4.2	Obfuscation	24
4.2.1	Obfuscation of expressions	24
4.2.2	Obfuscating constants	26
4.2.3	MBA Obfuscators	27

4.3	Mixed Boolean-Arithmetic (MBA) complexity	27
4.3.1	Incompatibility of operators	27
4.3.2	DAG representation	28
4.3.3	Complexity Metrics	29
4.4	Existing deobfuscation tools	31
4.4.1	Arybo	31
4.4.2	SSPAM	32
4.4.3	MBA-Blast	32
5	Design	35
5.1	Naive Approach	35
5.1.1	Varying Sliding Window Algorithm	36
5.1.2	Motivating example	37
5.1.3	Observations	40
5.2	Criteria	42
5.3	Classification Approach	42
5.3.1	Classification Pipeline	42
5.3.2	Constructing a classifier	43
6	Implementation	47
6.1	Data	47
6.1.1	Data Requirements	48
6.1.2	Existing Data Sources	48
6.1.3	Data Assessment	49
6.1.4	The Need For Another Dataset	51

6.1.5	Dataset Summary	52
6.2	Feature Extraction	53
6.2.1	Data Processing	53
6.2.2	Bag-of-instructions	53
6.2.3	Register Dependency N-Grams	54
6.2.4	Harmonizing features in a framework	56
6.3	Feature Analysis	56
6.3.1	Instruction amount	57
6.3.2	Token frequency	58
6.3.3	Dataset comparison	61
6.4	Model	62
6.4.1	Train-test split	62
6.4.2	Evaluation Metric	63
6.4.3	Naive Bayes	64
7	Evaluation	67
7.1	Evaluation Method	67
7.1.1	Evaluation Set	68
7.1.2	Segment labeling	68
7.1.3	Program Synthesis	69
7.2	Performance	70
7.2.1	SYNTIA and QSYNTH dataset	70
7.2.2	ALGO dataset	72
7.2.3	Entire dataset	73

<i>CONTENTS</i>	xix
7.3 Comparison with initial approach	74
8 Conclusion	75
Bibliography	77
Appendices	1

List of Figures

2.1	Illustration of the different steps in the development cycle and in reverse engineering. The blue arrows represent steps in the development cycle. The orange arrows represent steps used in reverse engineering.	6
2.2	Example of a control flow graph constructed in IDA Pro.	7
2.3	abstract syntax tree (AST) representation of a simple function [1].	8
2.4	abstract syntax tree (AST) representation of the expression $2 \times (x \wedge y) + (x \wedge y)$ [1].	8
2.5	Example of a simple opaque predicate transformation. The original code gets split by the opaque predicate, indicated in orange. The branch that is never taken leads to meaningless junk bytes to confuse the reverse engineer.	12
2.6	Example by László and Kiss [2] that illustrates the control flattening transformation.	13
2.7	Illustration by David et al. [3] showing the used simplification approach.	20
4.1	Directed Acyclic Graph (DAG) representation of the expression $2 \times (x \wedge y) + (x \wedge y)$ [1].	29
4.2	Directed Acyclic Graph (DAG) for $2 \times (x \wedge y) + (x \wedge y)$. The <i>bit-vector size</i> is added as attribute to the Directed Acyclic Graph (DAG), visible in subscript. . .	31
5.1	Visualisation of the <i>varying sliding window</i> algorithm for a MIN_WINDOW_SIZE of 1 and a MAX_WINDOW_SIZE of 3.	37
5.2	Analysis of time per iteration compared to the amount of instructions in that iteration.	41
5.3	Example of a <i>bag-of-instructions</i> for a part of an assembly code segment.	44

6.1	Illustration of the workflow for turning THE ALGORITHMS programs into useful binaries.	52
6.2	Example exercise for the construction of N-grams. We want to derive all bigrams and trigrams for the function that starts at address 8 and ends at address 24. . .	55
6.3	Illustration of how N-grams of a specific function are devised (here for N=3). First the entire program binary dependencies are mapped to a graph. Then the subgraph is taken that only contains the instructions of the function. Lastly, all the paths of length N are considered.	56
6.4	Boxplots of the amount of instructions per function for each dataset. Please note that the y-axis is not on the same scale for each plot.	58
6.5	Bar plots of the total frequency of every token compared to the amount of tokens in the dataset. Please note that the y-axis is not on the same scale for each plot.	59
6.6	Bar plots of the total frequency of every token compared to the amount of tokens in the dataset, with shortcuts instead of raw data dependencies.	60
6.7	Bar plots of the total frequency of every token compared to the amount of tokens in the dataset, with shortcuts instead of raw data dependencies and without push/pop dependency chains.	61
7.1	Visualisation of the different type of segments. The Mixed Boolean-Arithmetic (MBA)-obfuscated segment starts at obf1 and ends at obf4. The green blocks show one segment.	69
1	Bar plots of the average instruction frequency per instruction for each dataset. . .	4
2	Bar plots of the average bigram frequency per instruction for each dataset. . . .	5
3	Bar plots of the average trigram frequency per instruction for each dataset. . . .	6

List of Tables

5.1	Measurements of the naive approach on our motivating example	40
6.1	Overview of the class distribution for the different datasets after removing duplicates and keeping original and obfuscated functions in the same set.	63
6.2	Matthews Correlation Coefficient (MCC) scores for the multinomial naive bayes classifier using the token count as features. The rows indicate the dataset on which the model was trained and tested.	65
6.3	Accuracy scores for the multinomial naive bayes classifier using the token count as features. The rows indicate the dataset on which the model was trained and tested.	65
7.1	Amount of segments for each category, split by the different type of program binaries in the evaluation set.	69
7.2	Evaluation results of Naive Bayes classifier trained on the SYNTIA and QSYNTH dataset, using instructions as features.	71
7.3	Evaluation results of Naive Bayes classifier trained on the SYNTIA and QSYNTH dataset, using bigrams as features.	71
7.4	Evaluation results of Naive Bayes classifier trained on the SYNTIA and QSYNTH dataset, using trigrams as features.	72
7.5	Evaluation results of Naive Bayes classifier trained on the ALGO dataset, using instructions as features.	72
7.6	Evaluation results of Naive Bayes classifier trained on the ALGO dataset, using bigrams as features.	72

7.7	Evaluation results of Naive Bayes classifier trained on the ALGO dataset, using trigrams as features.	73
7.8	Evaluation results of Naive Bayes classifier trained on all datasets, using instructions as features.	73
7.9	Evaluation results of Naive Bayes classifier trained on all datasets, using bigrams as features.	73
7.10	Evaluation results of Naive Bayes classifier trained on all datasets, using trigrams as features.	74

List of Acronyms

ANF	algebraic normal form	27
AST	abstract syntax tree.....	7
CFG	control flow graph	7
DAG	Directed Acyclic Graph	28
DSE	dynamic symbolic execution	10
ISA	instruction set architecture	14
MATE	Man-At-The-End.....	2
MBA	Mixed Boolean-Arithmetic.....	13
MCC	Matthews Correlation Coefficient	63
MCTS	Monte Carlo Tree Search	17
NLP	natural language processing.....	43
SAT	boolean satisfiability	28
SMT	satisfiability modulo theories	10

1

Introduction

1.1 Reverse Engineering

Reverse engineering has been around since the beginning of civilisations. Throughout history, people have been reverse-engineering inventions to extract know-how and gain an edge in the world. This has often been the case in warfare; accounts date back to the Roman empire, where their victory in the First Punic War is attributed to them being able to capture and mass-produce a Carthaginian ship [4]. In modern times, the growth of software has created yet another field where reverse engineering plays a significant role.

Chikofsky and Cross define *reverse engineering* in software as “*the process of analysing a subject system to (i) identify the system’s components and their inter-relationships and (ii) create representations of the system in another form or at a higher level of abstraction*” [5]. This process has different use cases; it is needed when documenting legacy systems [6], or for the reimplementation of software on new hardware [7]. It is also crucial in the discovery and exploitation of security vulnerabilities. Understanding the inner workings of some software enables an attacker to make a program behave differently than intended, e.g. by crafting specific inputs that exploit design flaws or disabling certain security mechanisms within the code. Protecting against such attacks is not a trivial task, especially when the attacker has complete control over the environment in which the software runs. Such a scenario is called a Man-At-The-End attack.

1.2 Man-At-The-End Attack

In a Man-At-The-End (MATE) attack, the attacker has physical access to the execution platform and the software implementation. As a result, the attacker can control the execution environment, tamper with the software and hardware, analyse the software with various tools or even perform side-channel attacks. Due to this broad attack vector, traditional security solutions for remote attacks cannot be deployed in this scenario.

MATE attacks have their roots in the removal of licenses for programs [8]. However, the range of applications susceptible to MATE attacks has grown in parallel with the ever-increasing amount of digital services and systems. This increasing threat has sparked research regarding possible security measures. The majority of this research is focused on software protection [9]. It should be noted that malicious parties can also benefit from better software protection, given that malware can apply the same principles as legitimate software, making it tougher to detect and analyse [8]. Consequently, there is also research on how to break software protection. These two sides of research result in a cat-and-mouse game, where both protection and attack mechanisms keep evolving.

1.3 Thesis Outline

This thesis starts by exploring the existing MATE attack landscape in Chapter 2. We depict the broad software protection scene and more clearly define various obfuscation and deobfuscation techniques. Chapter 3 clarifies the goal of our work: the localisation of Mixed Boolean-Arithmetic code within a program binary. This is followed up by an in-depth analysis of Mixed Boolean-Arithmetic in Chapter 4. Consecutively, we expand on our methodology in Chapter 5, followed by the implementation in Chapter 6. We then evaluate this work in Chapter 7 and draw the conclusions and future work in Chapter 8.

2

Background

In this chapter, we provide some background to software protection. Section 2.1 clarifies what software protection is and why it is needed. Section 2.2 clarifies code-obfuscation and its wanted properties. In Section 2.3 and Section 2.4, we depict the methods for analysing binaries. This is followed up by obfuscation techniques in Section 2.5. Finally, Section 2.6 introduces the concept of program synthesis and its relevance for deobfuscation.

2.1 Software Protection

Software protection refers to mechanisms that try to protect software against reverse engineering, piracy and tampering [10]. They are used both by developers wanting to protect their code, as well as malicious parties trying to protect their malware from being detected and analysed. This section starts with the rationale behind software protection, followed by the classification of the types of protection.

2.1.1 Motivation

In a MATE attack, everything is under the control of the adversary. Hence, the attack vector in that scenario is only limited by the tools and knowledge of the attacker. For example, the attacker can attach a debugger to the program or alter the assembly during execution. He can measure power to perform side-channel attacks or tamper with the clock frequency to perform an instruction skip attack.

Since the skills and tools can evolve as new research surfaces, it is unlikely that deployed protection mechanisms hold up for an extended time. Therefore, a developer can only try to implement enough protection to disincentivise an attacker by increasing the effort needed for a successful attack.

For example, a game publisher tries to apply enough protections in order to generate as much revenue as possible before illegitimate copies appear. For the attacker, the incentive for cracking that game also decreases as time advances due to the diminishing public interest in said game.

Likewise, malicious parties try to obfuscate their malware to stay inconspicuous as long as possible and to delay defences against their work.

2.1.2 Categories

Software protection can be divided into four categories: code obfuscation, tamper-proofing, watermarking, and birthmarking [11]. Code obfuscation aims at making programs harder to reverse engineer. Tamper-proofing tries to make programs harder to modify by causing side-effects such as failure when running modified software. Watermarking tries to identify the program so that it can be tracked. Birthmarking is a way to detect if one's code is lifted from one program to another. The last three all benefit from obfuscation to complicate the detection and reversing of these mechanisms.

2.2 Code Obfuscation

Obfuscation techniques aim to make the program unintelligible as to complicate the reverse engineering process. An *obfuscator* \mathcal{O} can informally be described as a “compiler” that transforms a program \mathcal{P} into a new program $\mathcal{O}(\mathcal{P})$ that has the same functionality as \mathcal{P} yet is unintelligible to the attacker [12].

Multiple obfuscation transformations exist and, in order to complicate comprehension even

further, they can also be combined in a layered manner. However, obfuscation comes at a cost: obfuscation can expand program size, increase overhead, or even introduce new bugs. Hence, a programmer must consider the trade-off between the benefit of added protection against the drawbacks that accompanies this protection [13]. A common approach is to obfuscate only vital parts of the program with high-overhead techniques. The type of obfuscation then depends on the purpose of hiding that specific segment.

2.2.1 Metrics for obfuscation

The quality of an obfuscation is often defined by the following metrics [13, 14, 15]:

1. *Potency*: the amount of obscurity added by the obfuscation. A *potent* obfuscation increases the complexity of the program. It is hard to define this metric concretely. Intuitively, an obfuscation is potent whenever the protection does a good job at hiding the code's original intent to the attacker.
2. *Resilience*: how well a transformation holds up against human and automatic deobfuscation. The needed effort can, for example, be measured in time.
3. *Cost*: the additional memory, storage or time that the obfuscation brings with him.
4. *Stealth*: the measure in which an obfuscation can be detected. Ideally, obfuscated code is indistinguishable from the rest of the code.

By combining these four metrics, we can evaluate the quality of an obfuscation.

2.3 Program Analysis

Program analysis is essential for reverse engineering. By analysing the program, a reverse engineer tries to elevate his program understanding to a higher abstraction layer. This is the opposite of what happens during the development and compilation of a program. We illustrate this with Figure 2.1.

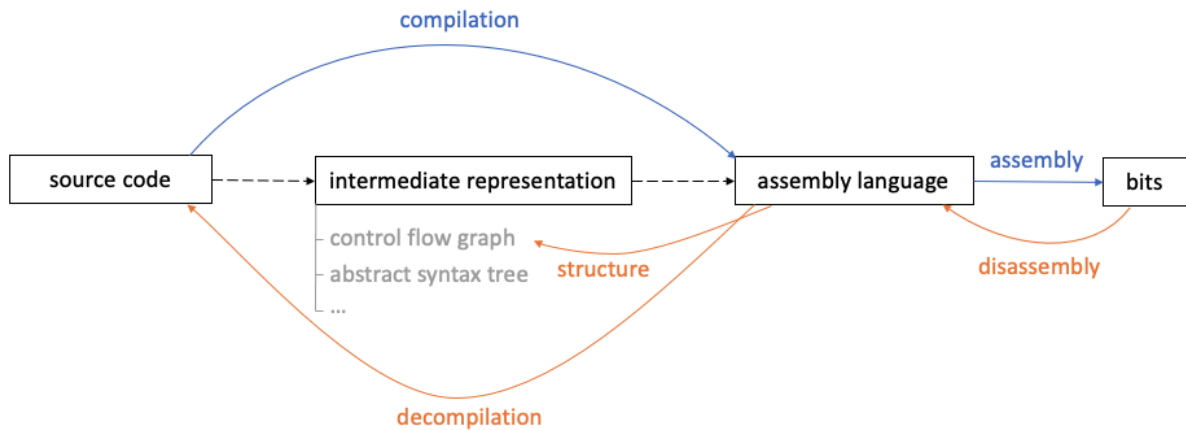


Figure 2.1: Illustration of the different steps in the development cycle and in reverse engineering. The blue arrows represent steps in the development cycle. The orange arrows represent steps used in reverse engineering.

In software development, a programmer writes code in a higher-level language, like C or C++. The compiler then compiles this code into assembly instructions. To do this, it uses various intermediate representations to optimise the program. The assembly then gets assembled into machine code.

The reverse engineer tries to reverse this process to get a better understanding of the control and data flow within the program. Intermediate representations can be very useful in this aspect. Contrarily to the forward process, the reverse engineering process does not have to be sound or complete. The only thing that matters is that the retrieved information provides insights about the program to the attacker.

Scripts and tools implement various techniques to analyse a target program. The different methods can roughly be divided into two categories: static analysis and dynamic analysis. Before explaining these categories in Section 2.3.2 and 2.3.3, we expand on two important graphical representations that assist the reverse engineer: control flow graphs and abstract syntax trees.

2.3.1 Graphical representations

Graphical representations prove to be very useful for a reverse engineer. They can reveal the control and data flow of the program and act as the basis for further analysis.

Control flow graphs

A control flow graph (CFG) represents all possible execution paths of a specific function [11]. Each function is divided into basic blocks; these are code segments (depicted in assembly) without any jumps. Edges between these basic blocks represent jumps in the control flow. A CFG reveals the internal flow of the program. As such, CFGs can reveal valuable information about the program's inner workings. An example of a CFG, constructed with IDA Pro [16] is shown in Figure 2.2.

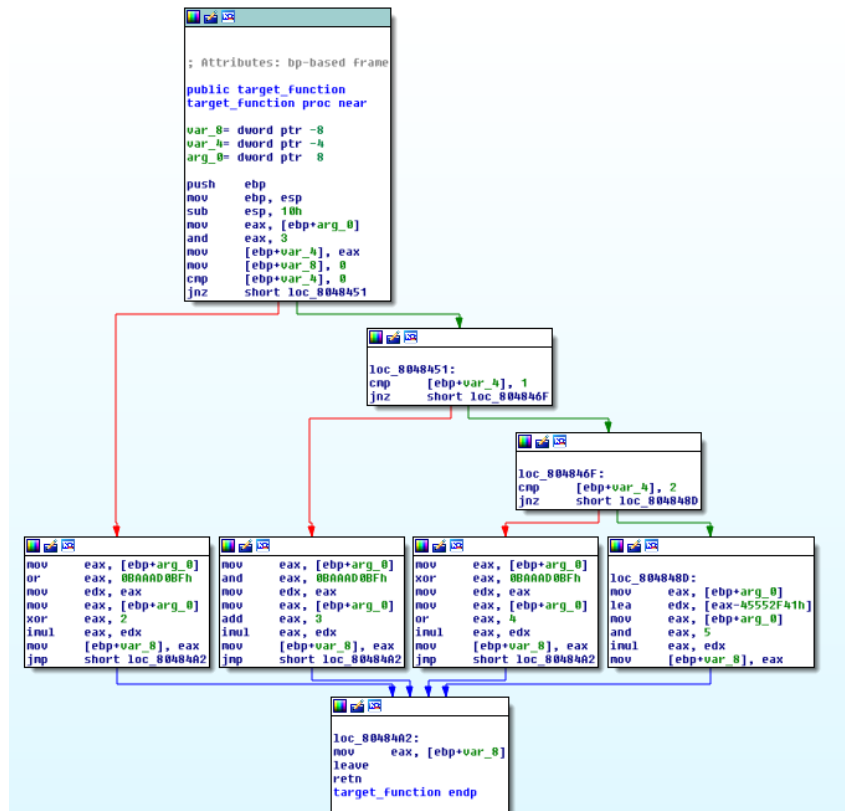


Figure 2.2: Example of a control flow graph constructed in IDA Pro.

Abstract syntax trees

Abstract syntax trees (ASTs) are often used to represent programs or expressions. Each node represents an operator, a function or a structure of the program. Each leaf usually corresponds to a variable or constant. ASTs can provide valuable insights into the control flow of a program or the data flow of an expression. An example of an AST for a function and for an expression are shown in Figure 2.3 and Figure 2.4 respectively.

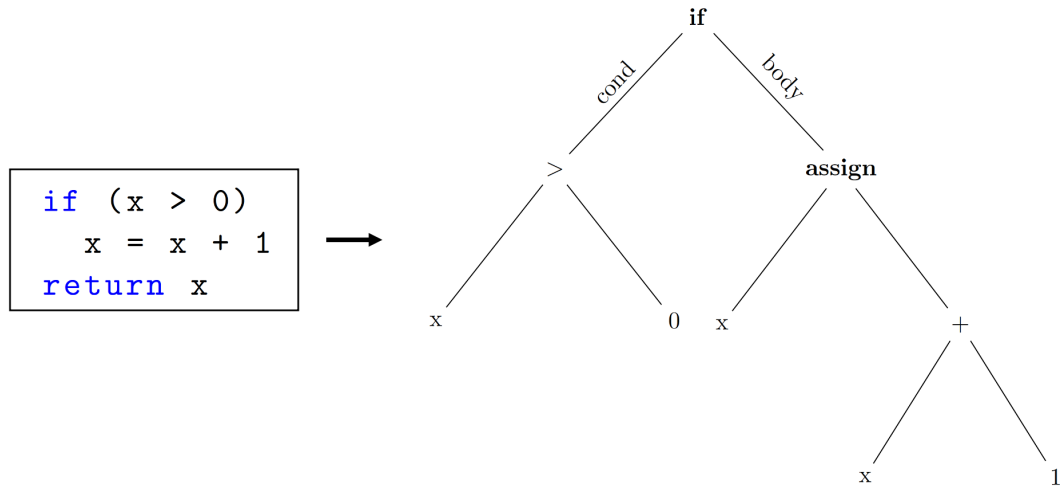
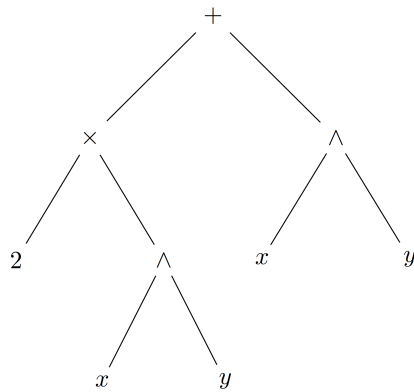


Figure 2.3: AST representation of a simple function [1].

Figure 2.4: AST representation of the expression $2 \times (x \wedge y) + (x \wedge y)$ [1].

2.3.2 Static analysis

In static analysis, all information is gathered without executing the code. Everything is inferred from the static application by inspecting the code and data as present in the binaries. The first step in this type of analysis is usually the disassembly of the machine language by a disassembler [17].

A disassembler tries to transform raw machine code into assembly code to make it interpretable by humans. The process is the opposite of that of an assembler used during the production of the application. Static disassemblers usually follow one of these two approaches: *linear sweep* or *recursive traversal* [17].

In *linear sweep*, the disassembler starts at the first byte of the binary's text segment and continues

from there, decoding one instruction after another. This technique is employed by GNU's `objdump` [18]. This approach suffers from potential errors due to embedded data between the instructions. In *recursive traversal* (used in IDA Pro [16]), the disassembler follows the control flow of the program. In this way, the disassembler avoids the problems with embedded data. However, the control flow can not always be reconstructed correctly, resulting in unexplored parts of the program. Therefore, a hybrid of the two approaches exist, called *speculative disassembly*, where unreachable code blocks are explored using *linear sweep*.

The assembly that is generated by the disassembler is not necessarily an exact reproduction of the original assembly. The quality of the generated assembly depends on the capability of the disassembler. For example, dynamically generated code and data (i.e. generated during the program's execution) will not be present in the disassembled code.

Another useful static analysis technique is the construction of control flow graphs, as explained in Section 2.3.1. After disassembly and control flow analysis using a CFG, decompilation can be attempted. This process is similar to disassembling, but the goal is a higher-level language, like C. As the compilation of a program removes a substantial amount of information, decompilation is not a trivial task. The chances of reconstructing meaningful source code are small. On top of this, the reconstructed code is often non-humanlike and thus difficult to understand [19].

An attacker can also resort to pattern matching. This technique tries to identify patterns of bits present in a database of implementations of some functionality. The attacker can construct these databases based on his previous manual work or by compiling known open source libraries.

Although static analysis can reveal valuable information, it can easily be thwarted by various obfuscation techniques. Static analysis also becomes less effective when working with sizeable programs, as it assumes no particular input or output and takes into account all possible executions of a program. Consequently, there is a need for an alternative approach: dynamic analysis.

2.3.3 Dynamic analysis

Contrarily to its static counterpart, dynamic analysis gathers information out of execution paths and data modifications by executing the program. These methods do not necessarily reveal all possible program execution paths, as the taken paths highly depend on the given inputs. However, the attacker is not necessarily interested in all those execution paths, as some will never be taken. This reduction can even be advantageous to simplify the problem, as some ignored edge cases might be unimportant.

A classic dynamic technique is debugging. Using a debugger, like GDB [20], an adversary can

pause the program, iteratively step through it, alter register and memory values and set breakpoints. This fine-grained control can quickly provide insights that would be very tedious to get in static analysis. It can also reveal dynamically created data and expressions. Certain tools, like PIN [21], take this concept one step further by enabling a user to insert new snippets of code in between the original program. This technique is called *instrumentation*. Dynamic analysis is also more robust against obfuscation techniques. Consequently, it plays an essential part in reverse engineering.

2.4 Symbolic Execution

Symbolic execution is a type of program analysis that explores the data flow of a program by using symbolic inputs instead of concrete ones [22]. In a traditional execution, a program runs based on specific inputs, leading to one specific control flow path. In contrast, symbolic execution uses symbolic values instead, enabling a *symbolic execution engine* to explore all possible control flow paths. For each path, the attacker can verify whether specific properties (e.g. null pointer dereferencing) are violated.

For each path, the *symbolic execution engine* keeps track of the branch conditions, under the form of a Boolean formula, and the mapping of variables to symbolic expressions or values. This information is then fed into *model checker* that checks if the wanted conditions are violated and verifies whether that path is realisable. The *model checker* is typically based on a satisfiability modulo theories (SMT) solver [23].

In traditional (static) symbolic execution, only symbolic variables are used. On top of the aforementioned limitations of static analysis, this method suffers from *state space explosion*. Therefore, it is only feasible for relatively small applications. Hence, a partial exploration of the space of possible execution states might be desirable. Contrarily to a complete exhaustive search, this method is not sound, but its performance gain is considerable.

A popular way to deal with *state space explosion* is to combine symbolic execution with concrete execution [24]. This is generally known as *dynamic symbolic execution (DSE)* or concolic execution [25]. In dynamic symbolic execution, the execution engine simultaneously executes the program concretely and symbolically. Whenever a branch arises, the symbolic execution can follow the path of the concrete execution, so the constraint solver does not need to evaluate whether that branch satisfies the wanted conditions. In order to explore multiple paths, different inputs can be used. This process can be repeated until the desired coverage is reached. DSE is commonly used in frameworks like Miasm [26] and Triton [27].

2.5 Obfuscation types

The point of obfuscations is to make the mentioned program analysis as arduous as possible. Collberg et al.[15] classify obfuscation transformations into three main classes, according to the kind of information they target: layout obfuscation, control-flow obfuscation, and data obfuscation. Layout obfuscation attempts to remove relevant information without changing behaviour. A typical example is the renaming of functions into meaningless characters [28]. Control-flow obfuscations aim to change the program's flow to make it more challenging to reconstruct abstract representations of the program, such as control flow graphs. Data obfuscation tries to transform application data and data structures, making them difficult to find or interpret.

With the purpose of staying ahead of newly developed deobfuscation methods, there is continuous research to invent new and enhance existing obfuscation techniques. Multiple research-oriented and commercial tools, such as Tigress [29] and Themida [30], offer a myriad of obfuscation transformations to programmers. Explaining and evaluating each of these transformations is out of the scope of this thesis. The following sections will outline some common data and control obfuscation techniques. For brevity, we will not expand on anti-tampering and anti-debugging techniques. For the interested reader, we refer to the study by Berlato et al. [31].

2.5.1 Opaque Predicates

A typical obfuscation for hampering control flow analysis is the insertion of opaque predicates. These are conditional expressions that will always evaluate to the same result [32]. This is known by the obfuscator but difficult to deduce by the attacker.

Opaque predicates can be constructed in different ways. For example, they can be based on a mathematical expression that always evaluates to true (or false). An example of this type is given in Figure 2.5. The obfuscator can then insert *junk bytes* in the path that is never taken. If the adversary does not notice the opaque predicate, he will consider this junk code when disassembling. This will result in an inaccurate reconstruction of the program behaviour, thereby complicating further control flow and data flow analysis.

There is broad research into different types of opaque predicates and ways to attack them. The example in Figure 2.5 is very basic. More complex constructions exist, like dynamic predicates that depend on each other or two-way predicates. These mechanisms are outside of the scope of this dissertation. For a broader overview of different types of opaque predicates, we refer to the work by Xu et al. [33]. Zobernig et al. also provide an extensive overview of the different attack strategies [34].

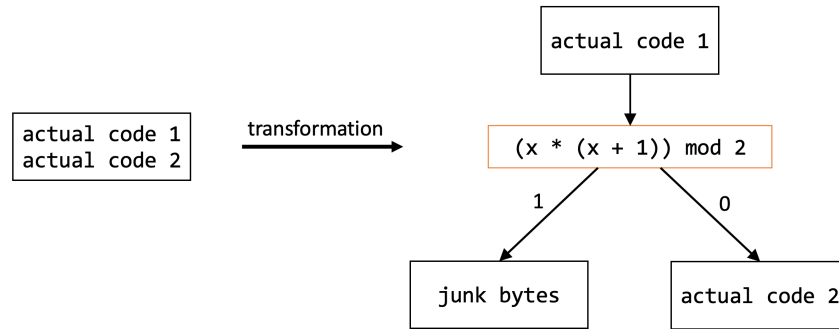


Figure 2.5: Example of a simple opaque predicate transformation. The original code gets split by the opaque predicate, indicated in orange. The branch that is never taken leads to meaningless junk bytes to confuse the reverse engineer.

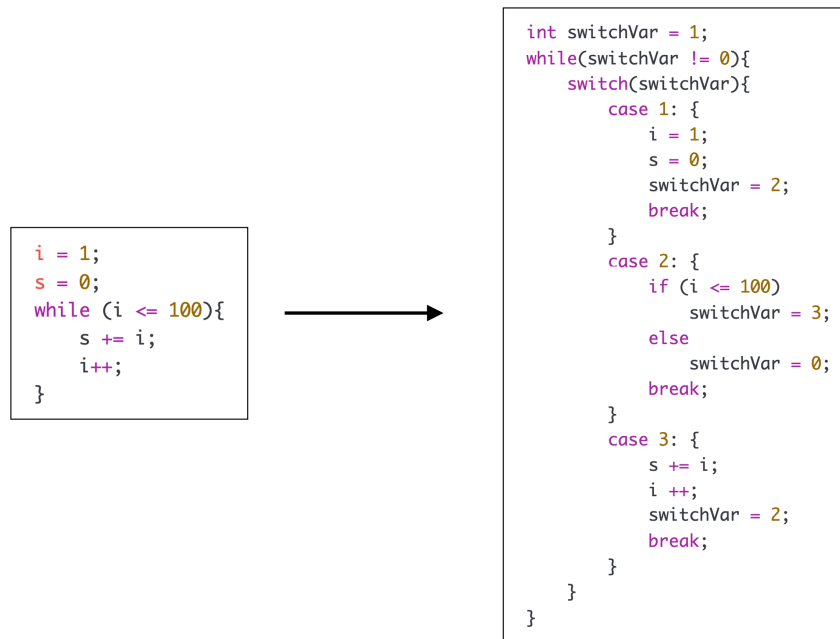
Please note that the predicate used in Figure 2.5 is merely an example to illustrate how opaque predicates work. The predicate in question is trivial to recognise and unobfuscate by the use of pattern matching.

2.5.2 Function inlining/outlining

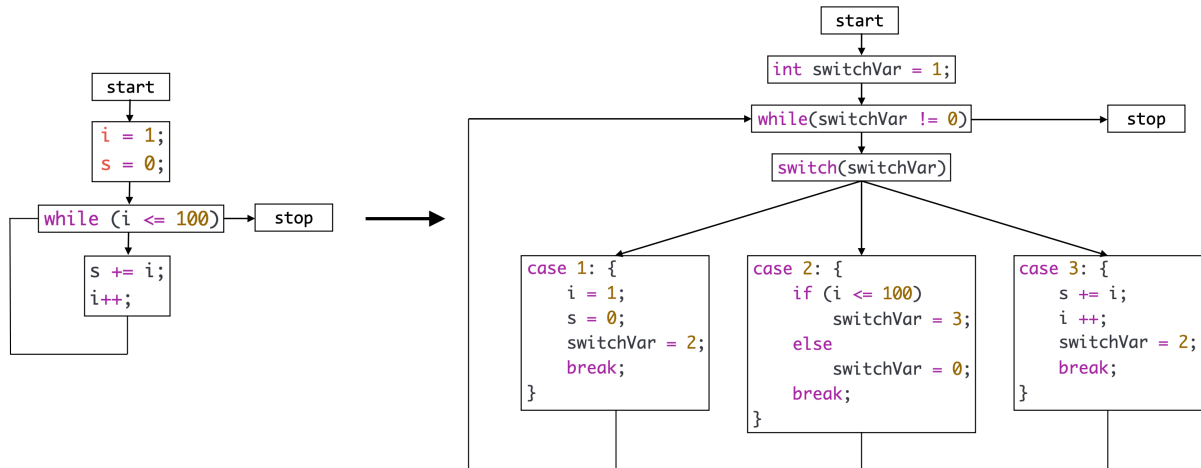
Function inlining and outlining is often done by compilers for optimisation [35]. Apart from potential performance enhancements, it can also be used in the context of obfuscation [36]. For *inlining*, a function call is replaced by the function body. For *outlining*, parts of a function are split as a separate function. Applying one of these techniques (or both) makes the control flow complicated, thereby confusing the attacker.

2.5.3 Control flow flattening

Flattening the control flow is another obfuscation that obstructs data and control flow analysis. With this technique, the branching between the different parts of a function are replaced by a call to a switch statement. This switch statement transfers control based on the value it receives. In this way, the targets of branches are difficult to determine with static analysis. Figure 2.6 illustrates the described transformation.



(a) Code before and after control flow flattening.



(b) CFG before and after control flow flattening.

Figure 2.6: Example by László and Kiss [2] that illustrates the control flattening transformation.

2.5.4 Mixed Boolean-Arithmetic

Since Zhou et al.[37] proposed this technique in 2007, it has been widely adopted to hide constants and simple expressions [38]. Mixed Boolean-Arithmetic (MBA) transforms these constants and expressions to a semantically equivalent but much more complex code.

In MBA, operators are transformed into an equivalent MBA expression and subsequently en-

coded into an expression that appears to have many more variables. This results in a complicated MBA expression, where the combination of classical arithmetic operators and boolean operators makes it hard to retrieve the original expression. We will further expand on this topic in Chapter 4.

2.5.5 Virtualisation obfuscation

In the context of obfuscation, virtualisation is used as a way to hide the control flow of the original program. So-called “virtualisation obfuscators” translate selected parts of the program machine code to bytecode in a new, custom virtual instruction set architecture (ISA) [39, 40]. This ISA is randomly generated at the time of protection. It is accompanied by a corresponding interpreter for this specific ISA. This interpreter is usually heavily obfuscated.

The virtualisation obfuscator stores the ISA in the target binary, along with the interpreter for this custom ISA. Every call to the original code snippet is replaced with the invocation of the interpreter. This results in an extra layer of abstraction that makes reverse engineering harder.

If an attacker wants to reverse engineer the program, he must first convert the created bytecode back into native machine code that resembles the original pre-protected code. To this end, he must try and understand the underlying architecture and instruction set. This process is highly time-consuming and is only specific to one ISA. As a result, virtualisation obfuscation is widely used in the industry. It is the basis of many obfuscation solutions, like VMProtect [41], Themida [30] and SecuROM [42].

2.6 Program Synthesis

A novel approach for program analysis is the adoption of program synthesis. Program synthesis is the task of automatically constructing a program based on a given high-level specification [43]. This technique is applied in different fields, ranging from business applications to program understanding.

In business applications, companies can leverage program synthesis can to assist non-programmer employees in automating repetitive tasks; By defining the intended behaviour, a program is automatically generated without requiring programming knowledge.

Program synthesis can also be used for program understanding. In the context of reverse engineering, program synthesis is the task of reconstructing an obfuscated code segment based solely on specific observations, like execution traces of input-output behaviour. The synthesiser tries to produce a program that retains the specified properties of the obfuscated code yet is more

readable for a reverse engineer. The result is a code segment that is easier to understand than the obfuscated program, essentially bypassing the effects of the obfuscation transformation.

Three main dimensions must be considered for the design of a program synthesiser [43]: the expression of the high-level specification, the search space and the search technique. These design decisions are explained in Section 2.6.1, 2.6.2 and 2.6.3 respectively, followed by some concrete implementations that are relevant to this thesis in section 2.6.4. For an in-depth overview of program synthesis and its many applications, we refer to the extensive study by Gulwani et al. [44].

2.6.1 Expressing the high-level specification

Multiple dimensions must be considered for the design of a program synthesiser [43]. First, the designer must consider the mechanism for the expression of the high-level specification. The particular choice depends on the given scenario. In reverse engineering, this mechanism is usually based on a trace of the obfuscated segment or on input-output behaviour.

2.6.2 Search Space

Next, the search space must be considered. This defines how the synthesiser reconstructs a candidate program. Not only should the synthesiser be able to synthesise the program segment, but the search space also needs to be constrained enough in order to obtain this segment in an acceptable amount of time. Therefore, the designer of the synthesiser needs to strike a good balance between the expressiveness and the efficiency of the search space. The search space is mainly defined by two attributes [44]:

1. **The operators used in the program:** for deobfuscation purposes, this is usually a combination of arithmetic operators (such as addition $+$, multiplication $*$, etc.) and bitwise operators (such as bitwise-and $\&$, bitwise xor \oplus , left-shift, etc.). More operators result in a larger search space.
2. **The control structure of the program:** this may e.g. be restricted to a bounded number of statements or to a loop-free program.

2.6.3 Search technique

The last important dimension is the search technique. The manner in which the search space is explored is a crucial factor in the effectiveness of the synthesiser. Various techniques exist,

based on enumerative search, deduction, constraint solving or stochastic techniques [44]. It is important to note that these techniques are not exclusive to each other. Different techniques are often applied together to form a concrete search to create a concrete search implementation.

Enumerative search

Enumerative search techniques generate a list of possible programs within the defined search space. In a later stage, this list is filtered on whether or not a program satisfies the given high-level specification. An enumerative search is a straightforward approach that, whilst simple, is very effective. The biggest constraint of this technique is scalability; too large a search space can result in a combinatoric explosion of the possible results. Therefore, a designer must carefully design the search space when implementing an enumerative search.

Deductive search

In *deductive* search, the synthesis problem is divided into smaller subproblems that can be seen as simpler synthesis problems. These small problems are solved separately in a divide-and-conquer way and combined to form the final synthesis.

Constraint Solving

The *constraint solving* technique breaks up the problem into two main parts: constraint generation and constraint resolution. In the first part, logical constraints (called the *synthesis constraint*) are generated that will yield the intended program. This is usually done by making some assumptions about the control flow of the program to be synthesised. In this way, the original program is transformed into constraints that the synthesiser can interpret (e.g. first-order logic). In the constraint resolution part, this *synthesis constraint* is then solved, usually by using existing SMT solvers. If a result satisfies all constraints, it corresponds to a valid reconstruction of the program.

Stochastic search

In *stochastic* techniques, the candidate program is viewed as a stochastic optimisation problem. This problem can be defined in different ways. The approaches to the stochastic techniques are outside of the scope of this thesis. Instead, the following section expands on the methods relevant to this thesis.

2.6.4 Concrete implementations

Brahma

In 2011, Gulwani et al. presented a technique for the synthesis of loop-free programs [45]. The synthesis is constructed as a combination of *components* from a given component library. These components describe the instruction set of the synthesised program, for example, a bitwise addition or an arithmetic shift. They are provided as logical relations between inputs and their respective outputs. Such an approach is called a *component-based synthesis problem*.

The researchers apply a constraint-based approach: they first reduce the problem of loop-free program synthesis to a satisfiability problem: finding the solution to a $\exists\forall$ first-order logic formula (the *synthesis constraint*). This formula encodes the space of all possible programs that can be obtained with the given components and satisfy certain well-defined constraints. Subsequently, this formula is solved by using a technique that leverages the power of modern off-the-shelf SMT solvers. The final result is a program that implements the wanted specification using only the components provided in the library.

Gulwani et al. implement this constraint generation and solving technique in a tool called BRAHMA. They test it on a dataset of 25 bit-manipulating programs of increasing complexity (1 to 16 bitwise operations) and two programs that also involve arithmetic. They demonstrate that BRAHMA can synthesise a solution for all these problems. It should be noted that this work is geared towards helping programmers code small bit-manipulating functions efficiently. Due to the parallels with reverse engineering, this tool can also be used for deobfuscation, albeit for only small fragments, especially when arithmetic is involved.

Syntia

At the 26th USENIX Security Symposium in 2017, Blazytko et al. proposed a new synthesis-based deobfuscation method and implemented it in a public prototype, called SYNTIA [46]. They employ a stochastic search technique by modelling the synthesis problem as a heuristic optimisation problem, where the search is guided by a Monte Carlo Tree Search (MCTS) based algorithm. They demonstrate that this enables them to successfully synthesise the high-level semantics of virtualisation obfuscation segments, mixed boolean-arithmetic segments, and ROP chains.

Their approach is divided into three parts: *trace dissection*, *random sampling* and *program synthesis*. These parts can be seen as steps in a pipeline. Each step takes the input of a previous step and provides an output for the next step. The process starts with an instruction trace of the

obfuscated segment. This trace is partitioned into unique sequences of assembly instructions, called *trace windows*. The goal of SYNTIA is to learn the high-level semantics of the different trace windows. These semantics are then provided to the reverse engineer, allowing him/her to gain insights and conduct further analysis.

Trace Dissection First, the boundaries of the *trace windows* need to be decided. This choice has a big impact on the synthesis result. If a *trace window* ends at an intermediary computation step, SYNTIA will not be able to construct a meaningful synthesis for that window. By default, the boundaries coincide with indirect control transfers. This heuristic corresponds to an invocation to the next handler for virtualisation obfuscation and a transition to the next gadget for ROP chains. An MBA expression is also usually not interrupted by an indirect control-flow transfer.

Random Sampling Once the different *trace windows* are defined, their semantics are captured by deriving input-output relations. First, SYNTIA identifies the inputs and the outputs of a trace window by use of the Unicorn Engine [47]. Both register and memory reads/writes are considered. A random sampler then generates random input values and derives the corresponding output values. This I/O behaviour is then fed into the program synthesis step. The tracing window itself is thus seen as a black box.

Program Synthesis The program synthesis tries to construct an expression that maps all the provided inputs to their corresponding outputs. This is known as *I/O oracle-guided program synthesis*. Blazytko et al. define a context-free grammar to formalise the search space for the synthesiser; all candidate programs are expressed in this grammar. A Monte Carlo Tree Search (MCTS) technique is used for the exploration of candidate nodes. This algorithm builds a search tree through reinforcement learning. Every new explored node is compared to previous nodes through a reward, calculated by a similarity function to the wanted input-output behaviour. The MCTS technique is combined with *simulated annealing* to try and escape local optima in order to end up with the best global solution.

By only using I/O samples to describe the semantics, the complexity of the synthesiser is not affected by the complexity of the obfuscated segment. Instead, the complexity of the synthesiser only depends on the semantic complexity of the underlying code. This is a significant improvement over BRAHMA, where extensive obfuscations would result in an arduous constraint that would be too difficult for modern SMT solvers.

Blazytko et al. evaluate their tool on a dataset of 500 randomly generated C functions, MBA-obfuscated through Tigress [29]. They were able to synthesise 448 out of the 500 functions in 34 minutes. They also tested SYNTIA on a program that was obfuscated with virtualisation, once with *VMProtect*[41] and once with *Themida*[30]. They managed to respectively synthesise 98% and 94% of the arithmetic and logical instruction handlers.

Qsynth

QSYNTH improves on SYNTIA by combining an input-output synthesis approach with dynamic symbolic execution [3]. The advantage of the input-output synthesis is that this technique is not hindered by the semantic complexity of the program. However, David et al. argue that this black-box approach results in a more difficult problem than the original deobfuscation. Namely, the size of the search space becomes too big to explore exhaustively. Hence heuristics are needed, like in the case of SYNTIA.

By combining input-output synthesis with dynamic symbolic execution, the semantics of the instructions can more accurately be modelled, drastically reducing the search space. The dynamic nature of DSE also bypasses certain obfuscations, like dead code or packing. Lastly, David et al. reduce the synthesis time even further by reusing results for similar obfuscated expressions.

The approach of David et al. consists of four steps. First, they trace the execution of the obfuscated program to obtain an execution trace. Then, they use DSE to compute a backward slice for the values of interest. With this slice, they construct an AST representation. This AST is then forwarded to the synthesis simplification algorithm. Lastly, the simplified expression is synthesised using a black-box offline enumerative synthesis oracle.

Program Tracing and DSE David et al. use a Dynamic Binary Instrumentation (DBI) to collect all instructions, together with their side-effects on registers and memory. Consequently, any obfuscated expressions that are not executed will not be included in this trace. DSE is now applied on this trace to obtain all symbolic execution paths, keeping track of symbolic values for both registers and expressions.

AST extraction The AST is obtained by using a backward slicing function. For the values of interest, this function first backtracks on logical control and data dependencies to model the data flow of the expression. This flow is then modelled as an AST, with nodes as operators and leaves as variables. This AST representation is used in all subsequent synthesis steps.

Offline Enumerative Search The synthesis approach is enumerative, meaning that all the potential expressions are enumerated. To this end, David et al. define a limited context-free grammar. They then exhaustively explore all the potential outcomes up to a certain depth. To prevent state space explosion, the synthesis uses a deductive element: instead of attempting to synthesise large expressions, the sub-expressions are first synthesised. On top of this, their approach can use precomputations: the enumerative search is only performed once. With these results, all the sub-expressions are synthesised. This is the *offline* part of the search. Lastly, they reduce the complexity of the search by using a constrained grammar, noting that a mapping of a larger grammar can be obtained with multiple mappings smaller subsets of the used grammar.

Simplification The researchers implement the deductive approach by first synthesising sub-expressions. This is illustrated in Figure 2.7. First, the leaf sub-expressions are replaced by a placeholder variable. This continues until the root is reached. Then, synthesis is applied in a random breadth-first fashion. Every time a subtree is successfully synthesised, it is replaced with a placeholder variable. This continues until all possible synthesis is done. In the end, the final AST is reconstructed by replacing all placeholder variables with their respective expression.

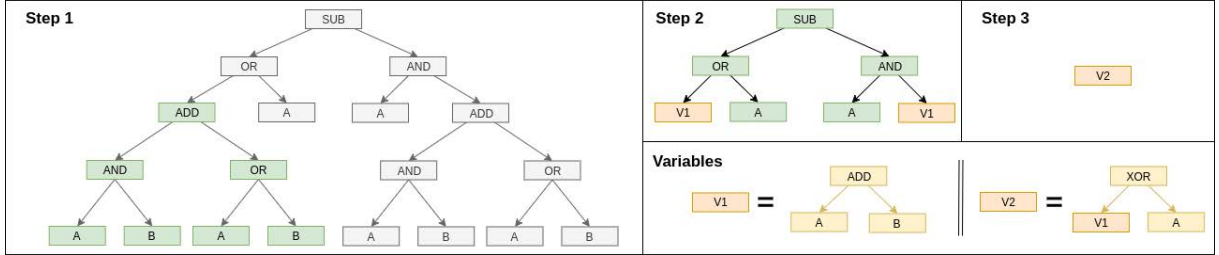


Figure 2.7: Illustration by David et al. [3] showing the used simplification approach.

David et al. implement this approach in a tool called Qsynth. They evaluate their approach on the MBA SYNTIA dataset and manage to synthesise 100% of functions in 1m35s, making this tool 20x faster than SYNTIA. They also evaluate their tool on a new dataset of 1500 more complex functions, obfuscated with MBA and virtualisation and manage to synthesise 82% completely. A more in-depth analysis can be found in their paper [3].

3

Thesis Objective

Program synthesis is a promising field for deobfuscating MBA and VM-based obfuscations. Their synthesis result provides insights about the semantics of the original code to the reverse-engineer. However, all the mentioned synthesisers assume that the reverse-engineer recognizes the use of obfuscations in a program binary and their exact location. Without this information, a synthesiser will not necessarily produce any meaningful result.

3.1 Thesis Goal

Intending to automate reverse engineering further, we aim to facilitate the manual process of locating and identifying obfuscations. More specifically, we look into automatically locating Mixed Boolean Arithmetic obfuscations in a given binary. Once an obfuscation is found, it can be fed to a synthesiser, like SYNTIA.

3.2 Thesis Scope

For this dissertation, we limit ourselves to the Intel x86 architecture. For the sake of consistency, all our experiments are performed on the same computer consisting of an AMD Ryzen

Threadripper 2990WX 32-Core CPU with a maximum clock speed of 3GHz, 64GB of RAM and a GeForce GTX TITAN GPU.

3.3 Thesis Outline

After the background study of Chapter 2, we now perform an in-depth analysis of Mixed Boolean-Arithmetic in Chapter 4. With this research in mind, we start on a design for our approach in Chapter 5. We then implement this design in Chapter 6 and evaluate it in Chapter 7. Lastly, the final conclusions are drawn in Chapter 8.

4

Mixed Boolean-Arithmetic

In this chapter, we give an in-depth analysis of MBA obfuscation. We start by defining MBA in Section 4.1. Section 4.2 explains how MBA is used in the context of obfuscation. Subsequently, the complexity of MBA expressions is then defined in Section 4.3. Lastly, the different deobfuscation efforts are applied in Section 4.4.

Apart from the initial papers by Zhou et al. [48, 37], there is barely any research on this topic. The most comprehensive work on mixed boolean-arithmetic is the PhD thesis by N. Eyerolles [1]. Because of the scarce literature, the majority of this chapter will be based on this work.

4.1 Polynomial Mixed Boolean-Arithmetic

In general, any expression consisting of both arithmetic operators ($+$, $*$, \dots) and boolean operators (and, xor, ...) can be referred to as a mixed boolean-arithmetic expression. Expressions of this form have already been used in other fields, like the design of cryptographic primitives. For this work, we specifically look at *polynomial mixed boolean-arithmetic* formalised by Zhou et al. [37] in the context of obfuscation.

Definition 1. (Polynomial MBA [37, 1]) *An expression E of the form*

$$E = \sum_{i \in I} a_i \left(\prod_{j \in J_i} e_{i,j}(x_0, \dots, x_{t-1}) \right)$$

where the arithmetic sum and product are modulo 2^n , a_i are constants in $\mathbb{Z}/2^n\mathbb{Z}$, $e_{i,j}$ are bitwise expressions of variables x_0, \dots, x_{t-1} in $\{0, 1\}^n$, $I \subset \mathbb{Z}$ and for all $i \in I$, $J_i \subset \mathbb{Z}$ are finite index sets, is a polynomial Mixed Boolean-Arithmetic (MBA) expression. A linear MBA expression is a polynomial MBA expression of the form

$$\sum_{i \in I} a_i e_i(x_0, \dots, x_{t-1})$$

Equation 4.1 is an example of a linear MBA expression that corresponds to $E = x + y$ [48]. Equation 4.2 shows a non-linear MBA expression [1].

$$E = (x \oplus y) + 2 \times (x \wedge y) \tag{4.1}$$

$$E = 85 * (x \vee y \wedge z)^3 + (xy \wedge x) + (xz)^2 \tag{4.2}$$

Notably, Definition 1 implies that the composition of polynomial MBA expressions is still a polynomial MBA expression: x_0, \dots, x_{t-1} can be polynomial MBA expressions of other variables. This ensures that we stay within the area of polynomial MBA expressions when combining them. For the sake of brevity, we will refer to *polynomial MBA expressions* as *MBA expressions* in the remainder of this thesis.

4.2 Obfuscation

MBA can be used for obfuscation in two ways. It can be used for the obfuscation of constants or for the obfuscation of expressions.

4.2.1 Obfuscation of expressions

Zhou et al. first formulated the technique of obfuscating expressions using MBA in 2006 [48] and formalised this in 2007 [37]. Since then, he has submitted several patents, together with other authors, based on this work [49, 50, 51]. Zhou proved that any bitwise expression can be transformed to a non-trivial linear MBA expression by iteratively applying one (or both) of the following transformations: subexpression rewriting and insertion of modular inverses.

Subexpression rewriting

In this transformation, parts of expressions are replaced by an MBA equivalent, chosen through the use of rewriting tables. To construct rewriting tables, Zhou et al. generate *MBA Zero functions*. This is an MBA expression that evaluates to zero irrespective of the values of the variables. They then isolate a small part of the zero function to be equal to the negative of the rest of the zero function. For example, out of the zero function in equation 4.4, we obtain the rewrite rule in equation 4.5.

$$-x - y + (x \oplus y) + 2 \times (x \wedge y) = 0 \quad (4.3)$$

$$\implies x + y = (x \oplus y) + 2 \times (x \wedge y) \quad (4.4)$$

$$x + y \rightarrow (x \oplus y) + 2 \times (x \wedge y) \quad (4.5)$$

For simplicity, we do not include the theory and proofs on how *zero functions* are constructed. This is explained in detail in the paper by Zhou et al. [37]. The essence of the method is that Zhou et al. find a zero function for the 1-bit space (i.e. for variables that are either 0 or 1) by using truth tables and are able to map this to integer space to be valid for any variable.

It should be noted that other MBA rewriting rules exist in the literature, like the *bit hacks* described by Warren [52].

Insertion of modular inverses

For this transformation: a part e of the expression is rewritten as $f(f^{-1}(e))$ with f an invertible function in $\mathbb{Z}/2^n\mathbb{Z}$. When $f^{-1}(e)$ is transformed with MBA rewriting, it becomes arduous for the attacker to retrieve e without knowing the exact function. f is usually an affine function as they are easily invertible and produce only a limited performance overhead. For 8-bit variables, an example of f is:

$$f : x \mapsto 39x + 23 \quad (4.6)$$

$$f^{-1} : x \mapsto 151x + 111 \quad (4.7)$$

The two transformations can be combined to obtain strong MBA expressions. For example, if we combine rewrite rule 4.5 with function 4.6, we can obtain the following MBA expression for

the variables $x, y \in \{0, 1\}^8$:

$$e_1 = (x \oplus y) + 2 \times (x \wedge y) \quad (4.8)$$

$$e_2 = e_1 \times 39 + 23 \quad (4.9)$$

$$E = (e_2 \times 151 + 111) \quad (4.10)$$

4.2.2 Obfuscating constants

Zhou et al. use the same principles of the expression obfuscation to obfuscate constants through MBA transformations [37]. For this, the following theorem is set forward:

Theorem 1. [1] *Let:*

- $P \in P_m(\mathbb{Z}/2^n\mathbb{Z})$ and Q its inverse: $P(Q(X)) = X \quad \forall X \in \mathbb{Z}/2^n\mathbb{Z}$,
- $K \in \mathbb{Z}/2^n\mathbb{Z}$ the constant to hide,
- E an MBA expression of variables $(x_1, \dots, x_t) \in (\mathbb{Z}/2^n\mathbb{Z})^t$ non-trivially equal to zero,

Then K can be replaced by $P(E + Q(K)) = P(Q(K)) = K$, no matter the values of the variables (x_1, \dots, x_t) .

For example, if we have a value K and zero function (4.4), we have:

$$K = P(x + y - (x \oplus y) - 2 \times (x \wedge y) + K)$$

The variables x and y can then be chosen randomly and the value of K will not be revealed. However, Eyrolles proves that this technique contains an algebraic weakness if the zero function contains no constant:

Lemma 1. [1] *Let:*

- $P(X) = a_0 + a_1X + a_2X^2 + \dots + a_dX^d$ be a polynomial of degree d ,
- Q be a polynomial of degree d , such that $P(Q(X)) = X$ for all $X \in \mathbb{Z}/2^n\mathbb{Z}$,
- $E = \sum a_i(\prod e_{i,j}(x_0, \dots, x_{t-1}))$ be a null MBA expression, with no $e_{i,j}$ such that $e_{i,j}(x_0, \dots, x_{t-1}) = 1$ whatever the values of x_0, \dots, x_{t-1} (i.e. the sum composing the MBA does not contain any constant).

Then the constant monomial of $P(E + Q(K))$ is equal to K .

Proof [1]:

$$\begin{aligned} P(E + Q(K)) &= a_0 + a_1(E + Q(K)) + \dots + a_d(E + Q(K))^d \\ &= a_0 + a_1Q(K) + a_2Q(K)^2 + \dots + a_dQ(K)^d + \varphi(E) \end{aligned}$$

with $\varphi(E) = \sum_{k=1}^d a_k \left(\sum_{i=0}^{k-1} E^{k-i} Q(K)^i \right)$, a polynomial in variables x_1, \dots, x_t with no constant monomial, as every monomial of $\varphi(E)$ is multiplied by a positive power of E . This means that the constant part of $P(E + Q(K))$ is:

$$a_0 + a_1Q(K) + a_2Q(K)^2 + \dots + a_dQ(K)^d = P(Q(K)) = K. \quad \square$$

Thus, zero MBA functions without a constant should be avoided as the value K can be revealed from the obfuscated form.

4.2.3 MBA Obfuscators

MBA is used in multiple commercial obfuscators, like Quarkslab [53] and Irdeto [54]. It is also available in the Tigress [29] obfuscator: expressions can be MBA-obfuscated using the *encode arithmetic* transform and data can be MBA-obfuscated using the *encode data* transform. For this thesis, we are mainly interested in the obfuscation of expressions. Hence, we refer to the *encode arithmetic* transformation when talking about Tigress in the remainder of this thesis.

4.3 MBA complexity

4.3.1 Incompatibility of operators

The strength of MBA obfuscations lies in the incompatibility between the arithmetic and bitwise operators. There are no general rules, like distributivity or associativity, for this kind of expression. This makes the simplification of these expressions arduous.

Canonical Forms

The advantage of purely arithmetic or purely boolean expressions is the existence of a canonical form. These forms represent a unique representation for any equivalent expression. Therefore, they are a perfect first step for simplification. Every equivalent expression is mapped to the same canonical form. It should be noted that these forms do not necessarily correspond to the most readable format for a human. We give an example of a canonical form for boolean functions: the algebraic normal form (ANF). Unfortunately, there is no canonical form for expressions that mix boolean and arithmetic operators.

Lemma 2. *Any n -bit boolean expression can be expressed in the following form:*

$$\bigoplus_{u \in \mathbb{F}_2^n} c_u \bigwedge_{i=0}^{n-1} x_i^{u_i}$$

where $c_u \in \mathbb{F}_2$, $x_i^{u_i} = x_i$ if $u_i = 1$ and $x_i^{u_i} = 1$ if $u_i = 0$, and \oplus is the bitwise XOR. This form is called the algebraic normal form.

Algebraic Tools

Common algebraic tools, like Maple [55], are almost purely geared towards arithmetic expressions. They provide limited functionality for boolean expressions. For example, Maple can do bitwise computations on constants only. These tools break down when provided with an MBA expression. They are therefore not usable for simplifying MBA expressions.

SMT Solvers

SMT solvers try to prove the satisfiability of logical formulas. They are a generalization of boolean satisfiability (SAT) problems by also supporting arithmetic, arrays, quantifiers and other first-order theories [56]. Popular SMT solvers, like Z3 [56], also offer a simplifier functionality. This simplifier applies standard algebraic reduction rules (e.g. $p \wedge \text{true} \mapsto p$), together with “limited contextual simplification”, for example $x = 4 \wedge q(x) \mapsto x = 4 \wedge q(4)$. Unfortunately, these simplifications do not provide a canonical form, and their simplified results are not necessarily more readable for a human. On top of this, contextual simplification, which is essential in MBA expressions, is limited. Therefore, SMT solvers are not suitable for deobfuscating MBA expressions.

4.3.2 DAG representation

To get a better sense of the complexity of an MBA expression, Eyrolles proposes a Directed Acyclic Graph (DAG) representation for MBA expressions. An example is given in Figure 4.1.

Definition 2. (DAG representation [1]) *The DAG representation of an MBA expression is an acyclic graph G where:*

- all leaves represent constant numbers or variables, other nodes represent arithmetic or bitwise operators;
- an edge from a node v to a node v' means v' is an operand of v ;

- *there is only one root node;*
- *common expressions are shared, which means they only appear once in the graph.*

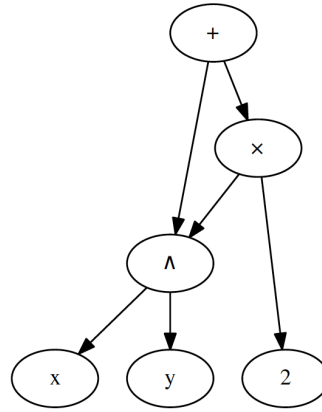


Figure 4.1: DAG representation of the expression $2 \times (x \wedge y) + (x \wedge y)$ [1].

4.3.3 Complexity Metrics

Eyrolles uses the DAG representation to define complexity metrics specific to MBA expressions [1]. Decreasing these metrics corresponds to a simpler MBA expression, both for human as for automatic analysis. While these metrics give a sense of complexity, they are not necessarily related to the resilience of the MBA obfuscation technique. For example, program synthesis does not consider semantic complexity when deobfuscating MBA expressions and is thus impartial to these metrics. On the other hand, the efficiency of traditional simplification tools (e.g. SMT solvers) are affected by these metrics.

Number of Nodes

The most evident metric is the *size* of the MBA expression. This can be expressed in the number of nodes in the DAG representation. The sharing property of the DAG representation plays an important role here: subexpressions that occur multiple times will not significantly increase complexity as they can be simplified in the same way. This is the case because every expression, obtained through the rewriting rules, has the same value irrespective of the value of the variables. The example in Figure 4.1 has a size of 6.

MBA Alternation

MBA alternation tries to capture the “MBA aspect” of the expression. A strong MBA expression needs to alternate between boolean and arithmetic operators. For example, if an MBA expression consists solely of arithmetic expressions, it can easily be simplified by any algebraic tool. In the same manner, subexpressions containing only arithmetic operands can be simplified within the MBA expression. Therefore, Eyrolles defines *MBA alternations* as the number of edges linking two nodes that represent operators of different types (boolean or arithmetic). In the example of Figure 4.1, the *MBA alternation* is 2.

Average Bit-Vector Size

The *bit-vector size* of a node refers to the number of bits of the variable that are truly important. We illustrate this with a small example: when combining a 32-bit variable with an 8-bit variable through an AND operand, an analyst only has to consider the eight least significant bits, as the others will be zero. So, the amount of important bits is only eight. Eyrolles summarises such simplification rules in the following definition of *bit-vector size*.

Definition 3. (Bit-vector size [1]). *For a node v , the bit-vector size $\text{bvsize}(v)$ is:*

- *If v is a leaf node, the bit size of the variable or constant it represents. This size can be deduced from the context (e.g. size of the input of a function), or by additional indications (e.g. binary masks). The size of a constant may also be inferred from the actual number of bits of that constant (possibly rounded to the next power of two).*
- *If v represents an operator, the bit size of the output of the operation. This depends on the nature of the operator:*
 - *if v represents a binary operator in $\{+, -, \times, \oplus, \vee\}$ with v_1, v_2 as operands, then $\text{bvsize}(v) = \max(\text{bvsize}(v_1), \text{bvsize}(v_2))$*
 - *if v represents a boolean AND with operands v_1, v_2 , then $\text{bvsize}(v) = \min(\text{bvsize}(v_1), \text{bvsize}(v_2))$*
 - *if v represents a unary operator in $\{\neg, -\}$, then $\text{bvsize}(v) = \text{bvsize}(v_1)$ for v_1 its operand.*

Using this definition, the *bit-vector size* can be computed for every node and added as attribute to the DAG representation. An average can then be taken over the entire DAG. Figure 4.2 shows the *bit-vector size* added in the DAG. In this case, the *average bit-vector size* is 22. This metric can be of use for deobfuscation approaches whose performance relies on this *bit-vector size*.

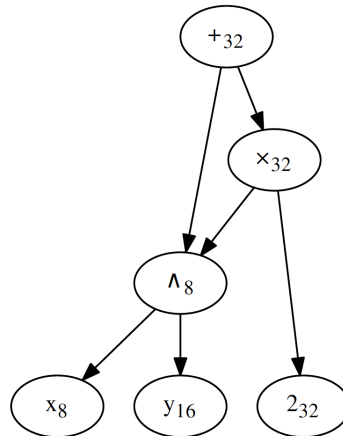


Figure 4.2: DAG for $2 \times (x \wedge y) + (x \wedge y)$. The *bit-vector size* is added as attribute to the DAG, visible in subscript.

4.4 Existing deobfuscation tools

Apart from the program synthesis techniques discussed in Section 2.6, the following paragraphs describe other efforts to deobfuscate MBA obfuscations.

4.4.1 Arybo

Guinet et al. propose a tool to derive a bit-level symbolic representation of MBA expressions [57]. This is known as a *bit-blasting* approach [1]. To obtain this representation, they translate the MBA expression into a bit-per-bit canonical form. They model arithmetic operators, like addition and multiplication, as a bit per bit boolean expression (e.g. using a full adder). They then derive the bit-per-bit algebraic normal form (ANF) of the entire MBA expression and attempt to translate it back to a word-level representation that is intuitive for humans (a word can be seen as any number of bits).

They devise a C++ library called `libpetanque` for the creation of the ANF form. This library stores the multiple bit vectors using the DAG representation described in Section 4.3.2. `libpetanque` transforms this DAG into an ANF canonical form. A python tool, called `ARYBO` then attempts to reconstruct a word-level representation of the ANFs. While the tool is successful in simplifying MBA expressions, its memory and time performance is directly related to the *average bit-vector size* of the MBA expression. For example, `ARYBO` can perfectly handle 8-bit expressions but fails to simplify 32-bit expressions in a reasonable time [1]. Guinet et al. also note that it is not easy to reconstruct a comprehensible word-level representation.

4.4.2 SSPAM

With SSPAM [58], Eyrolles et al. try to avoid the drawbacks of *bit-blasting* approaches like ARYBO. Instead of going down to the bit-level, they devise a simplification approach that stays on the word-level. The main observation is that the rewrite rules used for subexpression rewriting (described in Section 4.2.1) are reversible. Therefore, the attacker should be able to reverse these rewrites to obtain the original segment, provided that he knows the set of rewrite rules. However, the used rewrites are further complicated by the insertion of modular inverses. This must also be taken into account.

To identify the used rewrites, Eyrolles et al. resort to a pattern matching technique. They identify subexpressions that can be simplified to simpler expressions based on a database of rewrite rules. They construct these rewrite rules based on the method described by Zhou et al. [37] but also add support for the insertion of custom rewrite rules.

Apart from exact matching, SSPAM also implements *flexible* matching, which matches expressions that appear different but are equivalent. In this way, they try to get around the insertion of modular inverses. Lastly, the obtained expressions are further simplified using arithmetic simplification.

SSPAM proves to be effective at simplifying MBA expressions. The main downside of this method is the need for the original rewrite rules. With this research, Eyrolles et al. conclude that MBA-obfuscations that only use subexpression rewriting are quite weak. By also inserting modular inverses, side effects are introduced that are more effective at thwarting this simple pattern matching approach, making the MBA obfuscation more resilient.

4.4.3 MBA-Blast

At the 30th USENIX conference in august 2021, Liu et al. proposed MBA-BLAST [38]. They analyse the theory about expression rewriting in the paper by Zhou et al. [37] and conclude that there is also a mapping of integer space back to 1-bit space. This means that every MBA expression has a mapping to an MBA expression where all the variable values are only 0 or 1. Simplifying this expression is drastically simpler than solving an expression where all variables are integers.

Liu et al. provide mathematical proof for this finding and devise a tool that takes advantage of this property. MBA-BLAST essentially transforms MBA expressions to 1-bit space. It can then uses truth tables to enumerate all possible values and simplify the expression. The tool then remaps this expression to integer space to obtain the simplified expression.

Liu et al. test this method on a dataset of existing MBA expressions and MBA samples found in malware. They outperform all deobfuscation techniques mentioned above.

While this research seemingly renders MBA useless as an obfuscation technique, some assumptions seem to be missing in this paper. First of all, this approach only works on MBA obfuscations that rely solely on subexpression rewriting. It does not mention the effect of inserting modular inverses. MBA obfuscations without modular inverses were already proven to be weak by Eyrolles et al. in SSPAM [58]. Secondly, the proof provided by Liu et al. assumes that the MBA expression is linear. This means that the findings in MBA-BLAST are only applicable to a subset of MBA expressions. Nevertheless, the results of Liu et al. show that this tool is definitely valuable in practice. It manages to deobfuscate all MBA obfuscations found in malware.

5

Design

In this chapter, we present our general approach for the localisation of obfuscated mixed boolean-arithmetic code. In section 5.1, we start by applying a naive approach on an example to define the drawbacks and the shortcomings of the current workflow. Based on this, we define criteria for a better design in Section 5.2. We then attempt to construct a design that meets these criteria in Section 5.3.

5.1 Naive Approach

Our goal is to enhance the program synthesis pipeline by automatically locating MBA-obfuscated code in addition to the existing automatic synthesis. We start by creating a naive approach: applying program synthesis on every possible part of a binary and looking at the output. Section 5.1.1 describes a method for deriving all possible segments. We then apply this approach to an example and formulate some observations.

5.1.1 Varying Sliding Window Algorithm

To determine all possible segments, we devise an algorithm that is based on a *varying sliding window* approach. A pseudocode implementation is provided in Code Listing 5.1.1 and a visualisation is provided in Figure 5.1. Every iteration, we obtain a different window of the code segment of a program binary. The first window starts at the first instruction of the binary and is `MIN_WINDOW_SIZE` instructions long. The next window is one instruction longer than that. This incremental growth continues until the window has a size of `MAX_WINDOW_SIZE` instructions. At this point, the window moves an instruction and starts at the second instruction, again of size `MIN_WINDOW_SIZE` instructions. This window grows again until the maximum size is reached and moves again. This algorithm continues until the end of the code segment is reached.

Code Listing 5.1.1: Pythonic pseudocode for the naive *varying sliding window* approach

```

window_start = 0
while window_start < TOTAL_INSTR_AMOUNT-MIN_WINDOW_SIZE:
    # calculate range for end of current window
    min_window_end = min(window_start+MIN_WINDOW_SIZE,
        ↪ TOTAL_INSTR_AMOUNT)-1
    max_window_end = min(window_start+MAX_WINDOW_SIZE,
        ↪ TOTAL_INSTR_AMOUNT)-1
    # loop over all possible window sizes
    for window_end in range(min_window_end, max_window_end+1):
        # calculate window size in bytes
        byte_amount = addr_of(window_end)-addr_of(window_start)+1
        # feed window to Syntia
        solutions, time = syntia(addr_of(window_start), byte_amount,
            ↪ IO_AMOUNT)
        add_to_log(window_start, window_end, time, solutions)
    window_start += 1

```

The following equation shows the amount of segments depending on the variables:

$$\begin{aligned}
 \text{segmentamount} &= \sum_{n=\text{total}-\text{maxsize}+1}^{\text{total}-\text{minsize}+1} n \\
 &= \frac{1}{2}(1 + \text{maxsize} - \text{minsize})(2 * \text{total} - \text{minsize} - \text{maxsize} + 2)
 \end{aligned}$$

The amount thus grows linearly with the total amount of instructions, and quadratically with `MAX_WINDOW_SIZE` and the inverse of `MIN_WINDOW_SIZE`.

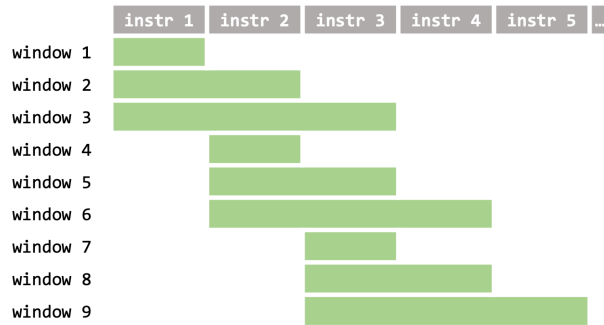


Figure 5.1: Visualisation of the *varying sliding window* algorithm for a `MIN_WINDOW_SIZE` of 1 and a `MAX_WINDOW_SIZE` of 3.

Every iteration, we pipe the obtained window to the SYNTIA program synthesizer. SYNTIA identifies the different inputs and outputs and generates a number `IO_AMOUNT` of input-output samples. Based on these I/O samples, it then attempts to synthesise an expression, using its MCTS-based heuristic as described in Section 2.6. When SYNTIA terminates, the synthesis results are stored in a log file and the next iteration starts. The log file also records the start and the end of each window together with the elapsed time for each corresponding iteration.

5.1.2 Motivating example

To get a grasp of the described workflow, we want to try out the described method on a simple compiled C program. Before applying the entire sliding window, we first test SYNTIA on a simple arithmetic C function. We can then incorporate this function in a program to analyze the entire approach. We opt to use SYNTIA [46] for our program synthesis approach as it is capable of deobfuscating MBA obfuscations and is publically available on Github [59].

Testing Syntia

Code Listing 5.1.2 shows a simple unobfuscated arithmetic C function. We compile this function using GCC [60] and feed the resulting binary to SYNTIA. SYNTIA perfectly manages to synthesize the given segment with the following result: `RAX = RDI+R8*RDX`.

Code Listing 5.1.2: Simple arithmetic function implemented in C.

```

int unobf_test (int v0, int v1, int v2, int v3, int v4){
    int r = (v0 + (v4 * v2));
    return r;
}

```

We proceed to obfuscate this arithmetic function using Tigress, leading to a mixed boolean-arithmetic function, shown in Code Listing 5.1.3. After compilation and some adjustments to SYNTIA (to fix bugs), we get the same synthesis results.

Code Listing 5.1.3: Same arithmetic function of Listing 5.1.2 but obfuscated using the Encode Arithmetic transformation in Tigress.

```

int64_t obf64_test(int64_t v0, int64_t v1, int64_t v2, int64_t v3,
↪ int64_t v4){
    int64_t r ;
    {
        r = (v0 | ((v4 & v2) * (v4 | v2) + (v4 & ~ v2) * (~ v4 & v2))) +
        ↪ (v0 & ((v4 & v2) * (v4 | v2) + (v4 & ~ v2) * (~ v4 & v2)));
        return (r);
    }
}

```

Running the naive algorithm

We now apply the *varying sliding window* algorithm on a small C program that contains the obfuscated function from Listing 5.1.3, shown in Code Listing 5.1.4. We compile this program using GCC to obtain the program binary and use `objdump` [18] to statically reconstruct the assembly instructions from the program binary. From this reconstruction, we derive the addresses of all instructions, that are then used in the *varying sliding window* algorithm to ensure that the start addresses of every window correspond with the first byte of an instruction and the end address corresponds with the last byte of an instruction.

Code Listing 5.1.4: Obfuscated arithmetic function of Listing 5.1.3 inside a working C program

```
int main(int argc, char** argv)
{
    int64_t v0 = atoi(argv[1]);
    int64_t v1 = atoi(argv[2]);
    int64_t v2 = atoi(argv[3]);
    int64_t v3 = atoi(argv[4]);
    int64_t v4 = atoi(argv[5]);

    int64_t r = (v0 | ((v4 & v2) * (v4 | v2) + (v4 & ~ v2) * (~ v4 & v2)))
    ↪ + (v0 & ((v4 & v2) * (v4 | v2) + (v4 & ~ v2) * (~ v4 & v2)));
    printf("%ld\n", r);
}
```

The entire binary contains 197 instructions. A minimum window size of 1 and a maximum window size of 197 would result in 19503 possible windows. In order to diminish this amount, we opt for a window size of at least 10 instructions and at most 50 instructions. We argue that this window size will be enough to encapsulate the obfuscation of this simple example. With these choices, there are 6888 candidate windows. For each window, we generate 20 input-output samples.

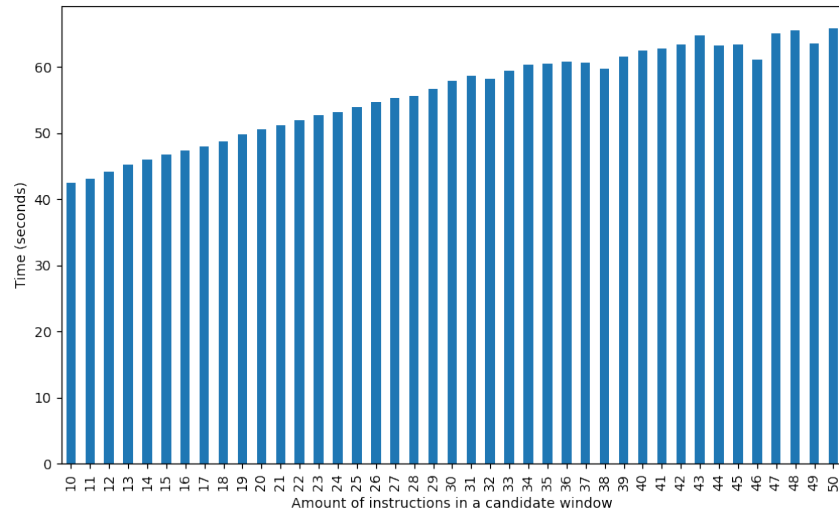
Table 5.1 summarizes the measurements of our experiment. The total time to run the naive algorithm on our example takes 63 hours. Of the 6888 candidate windows, 2685 candidates triggered an error in the random sampler due to bugs in the code, resulting in no possible synthesis. While the remaining 4203 windows return a successful synthesis, this synthesis does not always represent the obfuscated segment. As an example, results like `registerX=0` often appear when `registerX` remains unaltered in the analysed window. These synthesis results are correct, but they are of no use to the reverse-engineer. Out of all synthesised windows, only 32 windows reveal the behaviour of the obfuscated segment: `RAX = RDI+R8*RDY`.

candidate windows	6888
failed windows	2685
synthesized windows	4203
amount of correct results	32
total time	227022 seconds
average time per synthesized window	55 seconds

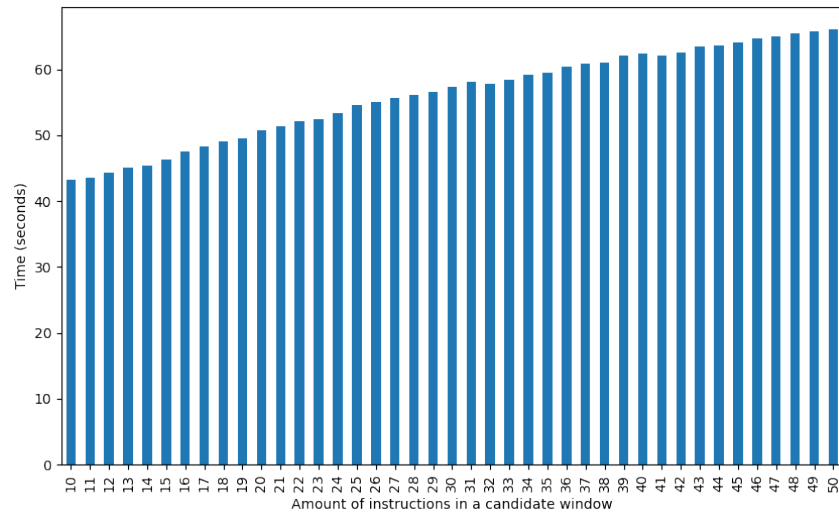
Table 5.1: Measurements of the naive approach on our motivating example

5.1.3 Observations

Based on the motivating example, we can draw three main observations. Firstly, one successful iteration takes an average of 55 seconds to complete on our system. This time increases as the candidate window gets larger, as seen in Figure 5.2. This means that it would take over 290 hours to compute all 19503 iterations, assuming a non-failing random sampler. We conclude that this naive approach scales badly, as it is already infeasible for this small, simple example.



(a) mean



(b) median

Figure 5.2: Analysis of time per iteration compared to the amount of instructions in that iteration.

Secondarily, it is not easy for a reverse-engineer to distinguish interesting synthesis results from the bulk of trivial or non-informative synthesis results. Most of the candidate windows contain multiple correct synthesis results that reconstruct the input-output behaviour of that window. However, most of these windows have no real meaning. These uninformative results create noise for the reverse-engineer. We could try to automatically filter out trivial results, like `registerX = 0`, but these can also be legitimate results, e.g. when using MBA to form an opaque predicate. Thus, the reverse-engineer has to manually filter out what looks interesting. For thousands of windows, this becomes time-consuming.

Lastly, out of the 6888 tested windows, only 32 recover the obfuscated segment semantics. These successful windows are all clustered around the obfuscated segment: this is the only place where the correct input and outputs can be observed.

5.2 Criteria

Based on the observations above, we draw the following criteria for an improved design. In order to get better scaling and reduced computation time, we want to limit the actual synthesis to windows of interest.

In the same manner, we want to cut down on the noise created by non-interesting synthesis results. Thus, we want to limit the use of SYNTIA by pinpointing the areas of interest before performing the synthesis.

There are relatively few windows where the correct result can be synthesised. This means that MBA windows must be accurately identified.

In summary, the criteria for a better approach at locating MBA code are:

1. Limit the number of times that SYNTIA is invoked in order to cut down on computation time and noise due to uninteresting results.
2. Be accurate enough in the identification of MBA windows in order to be able to recover the wanted semantics.

5.3 Classification Approach

To improve on the naive algorithm, we use a classification approach. We aim to design a classifier that can distinguish MBA windows from other windows. Section 5.3.1 details the resulting pipeline and the rationale behind this approach and Section 5.3.2 describes the construction of the classifier.

5.3.1 Classification Pipeline

Using a classifier to distinguish between MBA-obfuscated code and other code allows us to define a new workflow. This workflow can be seen as a pipeline: first, the program binary is analyzed to produce the instruction addresses. Based on this trace, all possible windows are derived in a

varying sliding window, as described before. The possible windows are then fed to the classifier that flags windows that likely contain MBA arithmetic. Now, only the flagged windows are fed to SYNTIA.

This approach drastically reduces the number of windows that are synthesised. As synthesis is the most time-intensive task, this reduction should significantly decrease the computation time of the *varying sliding window* approach. On top of this, fewer windows will contain synthesis results. Hence, the reverse-engineer will have to filter through less noise to find an accurate reconstruction of an obfuscated segment.

The time gain of this classification approach heavily depends on the time efficiency of the classifier and the accuracy of its predictions. The following section describes how such a classifier can be constructed.

5.3.2 Constructing a classifier

The classifier needs to be able to identify the use of MBA code based on the part of a program binary. We opt for a pattern-matching approach. Instead of devising concrete pattern matching rules, we leverage the power of supervised machine learning to infer rules from a dataset. This choice is based on the following observation: MBA obfuscation occurs before the compilation of the program. Hence, the optimization by the compiler will affect the MBA-obfuscated segment. On top of this, the assembly reconstruction of the program binary can also influence the resulting expression. Because of these two effects, patterns relying solely on the mathematical foundations of MBA would fail. As a bonus, a machine learning classifier is usually quite fast, as most of the time is spent during the training.

A supervised machine learning model tries to approximate the unknown function $y = f(\mathbf{x})$ based on a training set of example input-output pairs $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)$ [61]. The vector of inputs $x_1, x_2, \dots, x_k \in \mathbf{x}_i$ are called *features*, the outputs y_i are referred to as *labels*.

In our classification model, y_i is a discrete label that indicates whether the input corresponds to an MBA code segment or not: $y_i \in \{True, False\}$. The features \mathbf{x}_i are derived from a part of the program binary. The process of transforming raw data (program binary in our case) into features x_i is called *feature extraction*.

In the following paragraphs, we devise two feature extraction methods inspired by the field of natural language processing (NLP). Firstly, we adopt a *bag-of-words* method to be usable with assembly coding language. We then improve on this by defining *N-grams* in the context of assembly.

Bag-of-words

A traditional *bag-of-words* approach sees a text document as an unordered set of words with their position ignored [62]. Only the frequency of these words is kept. In the context of a program binary, we can use the assembly instructions instead of words. We will refer to this technique as a *bag-of-instructions*. For a given segment, every occurrence of each instruction is counted. The frequency of each instruction is then calculated and used as a feature for the machine learning model.

Take the example of a small part of a code segment, shown in Figure 5.3. Every instruction is counted and divided by the total amount of instructions. The frequency of each instruction corresponds to one feature x_i of the input vector.

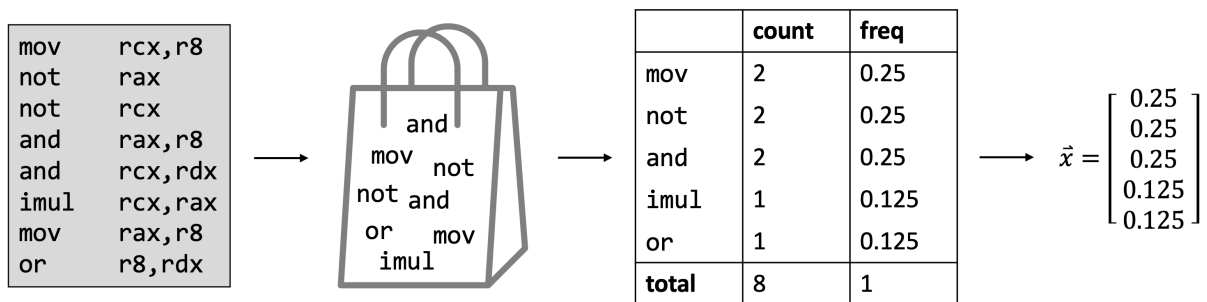


Figure 5.3: Example of a *bag-of-instructions* for a part of an assembly code segment.

A downside of the *bag-of-instructions* model is the fact that it takes no context into account. Traces with the same instructions but with a different order receive the same label. Therefore, we turn to *N-grams* instead.

N-grams

In natural language processing, *N-grams* try to encapsulate the order of words into machine learning features. An *N-gram* is a combination of N consecutive words. For text classification, the frequency of each *N-gram* in a text is registered and then used as a feature. Therefore, an *N-gram* can be seen as a generalisation of the *bag-of-words* scenario, with a *bag-of-words* corresponding to a *1-gram* or *unigram*.

Intuitively, information about the order of words is important for understanding a text segment: words that follow each other are often related in a sentence. In assembly code, this is not necessarily the case. Consecutive instructions can act on different registers and memory addresses, being placed in a particular order because of compiler optimisations.

Instead of creating *N-grams* based solely on the instruction order, we also take register dependencies into account. Instructions that form a read-after-write dependency chain can form an *N-gram*. As MBA expressions mix boolean and arithmetic instructions, their *N-grams* should be distinguishable from ordinary *N-grams*. Just like in natural language processing research, we will limit ourselves to 2-grams and 3-grams (often called digrams and trigrams). Larger N-grams result in a bigger and sparser featureset, both attributes that will lead to overfitting and worse performing models.

6

Implementation

In this chapter, we discuss the software implementation of our proposed designs. The entire codebase can be found in our Github repository [63]. We start with an analysis of available data sources in Section 6.1, along with the motivation for creating a new dataset. Section 6.2 outlines the methodology for obtaining the features designed in Chapter 5. This is followed by an analysis of these features in Section 6.3. Finally, Section 6.4 discusses the choice and implementation of the model for our classifier.

6.1 Data

When constructing a machine learning model, it is essential to use enough training data. Instead of mindlessly gathering data sources, it is important to define requirements for the data, tailored to the purpose of the machine learning task. We start by outlining these requirements in Subsection 6.1.1. Subsection 6.1.2 then explores the potential sources of existing data, and Subsection 6.1.3 assesses their utility for the problem at hand. In Subsection 6.1.4, a new dataset is put forward.

6.1.1 Data Requirements

Our goal is to locate MBA-obfuscated code in a program binary. Thus, the training data must consist of program binaries that contain MBA obfuscations. To train our models, we need to provide the location of the obfuscated segments in the program binary.

Manually labelling obfuscated segments of numerous program binaries is infeasible. Instead, we need to find a way to label specific segments as obfuscated/unobfuscated automatically. Instead of considering any segment, we opt to look at entire functions as obfuscated/unobfuscated. Functions are easily identifiable in (non-stripped) binaries. On top of this, obfuscation tools, like Tigress, also apply obfuscations on a per-function basis. With this knowledge, we can automatically label functions in program binaries, knowing that they are either one obfuscated segment or at least contain an obfuscated segment.

This choice results in the following requirements for the used program binaries:

1. The program binary may not be stripped, as the debug information is crucial to locating the different functions.
2. Functions may not be inlined. Else, a function could be labelled as unobfuscated while containing the inline code of an obfuscated function.
3. Ideally, an unobfuscated variant of the function should also be present. This ensures that our model contains a healthy mix of obfuscated and unobfuscated functions.
4. Preferably, the program binaries are compiled with the GCC `-O2` flag to reflect real-world binaries.

With these data requirements in mind, we start to look for existing datasets of MBA-obfuscated program binaries.

6.1.2 Existing Data Sources

We attempt to acquire data by looking at other research in the field. Fortunately, both the researchers of the SYNTIA framework and QSYNTH have devised a dataset to evaluate the performance of their synthesis approaches.

Syntia

As stated in their paper, Blazytko et al. created a dataset of 500 MBA-obfuscated functions [46]. They randomly generate 500 C functions and encode them with Tigress arithmetic encoding, followed by the Tigress data encoding. The result is a collection of 500 binary files. Unfortunately, the SYNTIA repository only contains said binary files without debug information. The source C files are also missing from the repository. Hence, we do not possess enough information about the program binaries to derive meaningful training data for our model.

QSynth

While the QSYNTH tool itself is closed source, the used datasets are available on Github [64]. The repository consists of four datasets. Notably, the researchers provide a complete reconstruction of the SYNTIA dataset. They include the unobfuscated C functions, the obfuscation script and the resulting obfuscated C functions, along with the program binaries. With these files, we have enough information to label the SYNTIA dataset for our research. The presence of the C files also allows for analysis of the data quality.

Apart from the SYNTIA dataset, David et al. construct three other datasets:

1. **EA**: 500 C functions obfuscated with the Tigress arithmetic encoding.
2. **VR_EA**: 500 C functions obfuscated with the Tigress virtualization transformation, followed by the Tigress arithmetic encoding.
3. **EA_ED**: 500 C functions obfuscated with the Tigress arithmetic encoding, followed by the Tigress data encoding.

Similar to the recreated SYNTIA dataset, the researchers provide the unobfuscated C functions, the obfuscation script and the resulting obfuscated C functions along with the program binaries. Upon closer inspection, the three new datasets are all transformations of the same unobfuscated C functions. Only the type of obfuscation differs. We are mostly interested in the arithmetic encoding and will thus use the **EA** dataset in the remainder of this thesis.

6.1.3 Data Assessment

Code Listing 6.1.1 showcases an example of an unobfuscated C function of the SYNTIA and the QSYNTH dataset. Both datasets were automatically generated. They were devised to evaluate the performance of program synthesis techniques. While they are useful in that regard,

they do not necessarily represent real-life functions where MBA obfuscations would be applied. Therefore, their usefulness is limited when trying to extend findings to real-world binaries.

Nonetheless, these functions can still be of use for our problem. Every function only contains one expression that needs to be obfuscated. Thus, the function can be seen as one obfuscated segment. In comparison, real-world functions often contain extra logic (e.g. for logging or output) that is not obfuscated. Such functions would result in noisy data when labelled as obfuscated, as some parts are not. So, thanks to the simplicity of the functions, we can ensure that obfuscated functions are entirely obfuscated. This results in clean data for our model.

Code Listing 6.1.1: Example of an original unobfuscated C function of the SYNTIA dataset and the QSYNTH dataset

```
uint64_t syntia_unobfuscated1(uint64_t a, uint64_t b, uint64_t c,
↪ uint64_t d, uint64_t e){
    uint64_t r = ((e & b) * d);
    return r;
}

uint64_t qsynth_unobfuscated1(uint64_t a, uint64_t b, uint64_t c,
↪ uint64_t d, uint64_t e){
    uint64_t r (((b-(b&b))&b)*(b^e))+(((b^e)^b)*(e&e)));
    return r;
}
```

When examining the original C files of both datasets, we notice a difference in how both sets are devised. Both datasets implement functions that mix arithmetic operators (+, *, /) with bitwise operators (AND, NOT, XOR, OR). However, the complexity of these functions differs significantly between the two. The SYNTIA dataset consists of functions with at most three variables and a maximum expression depth of 3. The QSYNTH functions have at most three variables, but their instruction depth is significantly larger. On top of this, some of the QSYNTH functions are complex representations of simpler functions. As an example, the QSYNTH function in Code Listing 6.1.1 is equivalent to calculating $e * e$. Hence, we could interpret the unobfuscated expression as obfuscated already. It is improbable that a programmer would implement such a function manually. For this reason, we argue that the SYNTIA dataset is more representative of actual unobfuscated code. Due to the limited amount of data, we still opt to use both datasets for our experiments.

6.1.4 The Need For Another Dataset

In the previous subsection, we hypothesised that the SYNTIA and QSYNTH datasets are usable for our problem, despite not being representative of real-life functions. To attest or refute this claim, we require a more realistic dataset comprising programs that serve a purpose other than benchmarking. We plan to use meaningful C programs, obfuscate them using the Tigress encode arithmetic transform, and compile them with `GCC -O2` to get usable program binaries. In this way, we can generate a realistic dataset. The following paragraphs investigate some possible sources of interesting C programs.

OpenSSL/TLS

The obvious choice when it comes to a realistic dataset is the OpenSSL/TLS Toolkit [65]. It is a popular open-source toolkit for cryptography and SSL/TLS applications. Some examples include the generation of public-private keys and the generation of TLS certificates. Whilst OpenSSL/TLS aims to be completely transparent, cryptographic programs of a similar kind might want to obscure their inner workings. Therefore, we plan to alter the OpenSSL/TLS framework by obfuscating certain functions to obtain an obfuscated program binary of the framework.

After figuring out how to manually build the toolkit, we are unfortunately stopped by the capabilities of the Tigress obfuscator. As OpenSSL/TLS is an extensive toolkit, it contains many C files, header files and an extensive build process. However, Tigress only accepts programs that consist of one C file [66]. The reason is that Tigress needs to do whole-program analysis and transformations. The makers of Tigress propose to merge the project into one C file, but this is infeasible for such a big project. Consequently, we decide to abandon the OpenSSL/TLS Toolkit as a data source and focus on smaller projects instead.

The Algorithms

With larger projects out of the picture, we turn towards the broader internet in search of usable programs. We finally land on THE ALGORITHMS [67], a collection of Github repositories that contain implementations of various computer science algorithms in multiple different coding languages. The repository for the C language contains 308 C files. Almost all files contain one standalone C program. The implemented algorithms range from number conversions to data structures to hash computations. We decide to use this mix of programs as a representative dataset. We will abbreviate THE ALGORITHMS as ALGO in the remainder of this thesis for conciseness. The next paragraphs describe how we turn this list of unobfuscated C programs

into usable program binaries.

Figure 6.1 portrays the workflow for producing obfuscated binaries based on an ALGO program. Before obfuscating the program, the target function to obfuscate must be defined. To retrieve all the functions in the 308 different C files, each file is first compiled using `GCC`. Then, the assembly of the resulting binary file is reconstructed using `objdump`. A script then derives the different function names by using regular expressions. These function names are then fed to Tigress, which creates the corresponding obfuscated C files and uses `GCC` to create the obfuscated binary simultaneously. After this process, a binary of the original file and a binary per obfuscated function is available. Hence, our dataset is created.

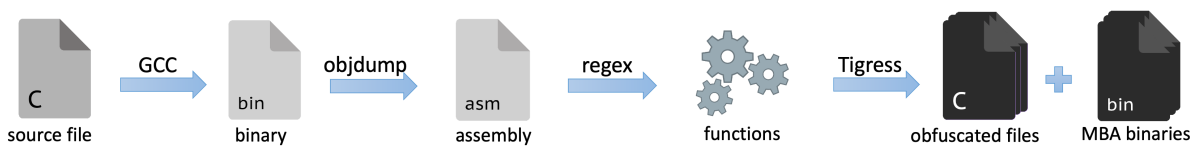


Figure 6.1: Illustration of the workflow for turning THE ALGORITHMS programs into useful binaries.

After applying this workflow on all the ALGO programs, 143 programs remain. The others either failed to compile with `GCC` or failed during the obfuscation by Tigress. We now analyse the obfuscated functions to remove the ones that are wrongly labelled as obfuscated. This happens when Tigress tries to transform a function without arithmetic. The result is exactly the same function. After this filtering, we end up with 400 obfuscated MBA functions and 2075 unobfuscated functions.

6.1.5 Dataset Summary

After this data gathering and creation process, the entire dataset consists of the following data:

- The SYNTIA dataset, consisting of 500 unobfuscated simple benchmark functions and their 500 MBA-obfuscated counterparts.
- The QSYNTH dataset, consisting of 500 unobfuscated simple benchmark functions and their 500 MBA-obfuscated counterparts.
- The ALGO dataset, consisting of 2075 unobfuscated functions and 400 MBA-obfuscated counterparts. These functions are from real programs and should be more truthful to real-life binaries.

6.2 Feature Extraction

Currently, all the data is provided in the form of program binaries containing (un)obfuscated functions. These binaries need to be streamlined into useful features for machine learning models, in accordance with the design choices of Chapter 5. Subsection 6.2.1 describes the general data processing, followed by the particular steps to obtain the *bag-of-instructions* and the *register dependency N-grams* in Subsection 6.2.2 and Subsection 6.2.3.

6.2.1 Data Processing

Before deriving the actual features of the data, the program binaries are combined into a list of functions and their attributes. For each function, the following attributes are stored:

1. The name of the program binary
2. The start address of the function
3. The name of the function
4. The instructions inside the function
5. The address of the instructions
6. A label indicating whether this function is obfuscated using MBA or not.

This information is statically inferred by applying custom regex filters on the assembly reconstruction of the program binary, obtained through `objdump`. The information is stored inside a `csv` file. This file can now be loaded as a `pandas` dataframe and altered into the wanted features.

6.2.2 Bag-of-instructions

The implementation of the *bag-of-instructions* is relatively straightforward. First, the entire dataset is traversed to obtain all the different instructions. A specific order is defined that will be used to express the frequency of each instruction in a function. For example, if there are three instructions in total, the order is: `[AND, OR, XOR]`. If a function contains 3 times `AND` and 2 times `XOR`, the corresponding instruction frequency will be expressed in a list as `[0.6, 0, 0.4]`. This list forms the features for our classifier.

6.2.3 Register Dependency N-Grams

Register dependency N-Grams list all read-after-write register dependency chains of length N for a certain function. To achieve this, we must first derive all register dependencies inside the program binary. We can then proceed to construct the available N-grams.

Deriving register dependencies

We use PIN to derive the register dependencies within a program binary [21]. PIN is a dynamic binary instrumentation tool by Intel. It performs instrumentation at run time on compiled program binaries by the use of *pintools*. A *pintool* defines the instrumentation process. This can vary depending on the instrumentation purpose. In our case, we are interested in the register dependencies. To this end, we use a data dependency pintool, developed in CSL, the computer systems lab where this thesis is conducted. For each instruction, the pintool collects the other instructions it depends on, both through memory as through registers. The output is a `csv` file that contains instruction address pairs, together with what register/memory linked them. Code Listing 6.2.1 shows an example of the output.

Code Listing 6.2.1: Example output after running the data dependency /textitpintool

Write_off addr,	Register,	Read_off addr
4128	, rsp	, 4239
4128	, rsp	, 18149
4150	, rax	, 4128
4166	, rcx	, 4128

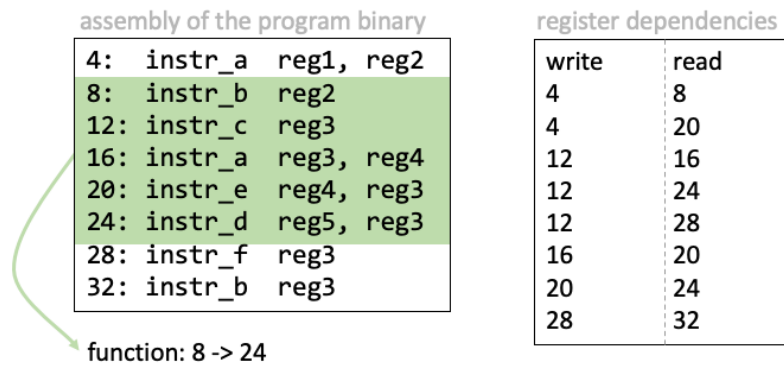
When running the pintool, we run into a major issue: to retrieve the register dependencies, the pintool dynamically runs the program binaries. So, the pintool will only capture data dependencies for parts of the program that are actually executed. This means that we need to provide the correct inputs for a program to work. If these inputs are not provided, the program terminates with an error message and without performing any computations. In this case, the data dependencies of most instructions are never captured.

The inputs for the SYNTIA and QSYNTH datasets are straightforward: all the functions require five input numbers to return a computed result. However, for the ALGO dataset, every algorithm expects different inputs. This requires an extensive manual process for all 143 programs. Due to time constraints, we defined the inputs for 30 programs. This cuts down the usable ALGO dataset to 96 obfuscated functions and 599 unobfuscated functions.

Constructing N-Grams

All the data is now available to construct N-grams for every function in a dataset. On the one hand, we have a list of register dependencies between every two instruction addresses in the program binary. On the other hand, we know for each function which instruction addresses it contains. These two pieces now need to be combined to form all the N-grams of a specific function.

Figure 6.2 shows an example to illustrate the complexity of the problem at hand. The assembly of the program binary, along with the register dependencies of this binary are given. Suppose that we want to derive all bigrams and trigrams of a function that starts at address 8 and ends at address 24.



assembly of the program binary		register dependencies	
4:	instr_a reg1, reg2	write	read
8:	instr_b reg2	4	8
12:	instr_c reg3	4	20
16:	instr_a reg3, reg4	12	16
20:	instr_e reg4, reg3	12	24
24:	instr_d reg5, reg3	12	28
28:	instr_f reg3	16	20
32:	instr_b reg3	20	24
		28	32

function: 8 -> 24

Figure 6.2: Example exercise for the construction of N-grams. We want to derive all bigrams and trigrams for the function that starts at address 8 and ends at address 24.

The bigrams can be obtained by considering every pair of the register dependencies and only keeping the pairs where the addresses are inside the wanted function. For the given example, this results in the following pairs: (12,16) (12,24) (16,20) (20,24). After translating these addresses to instructions, the resulting bigrams are [(instr_c, instr_a) (instr_c, instr_d) (instr_a, instr_e) (instr_e, instr_d)].

To obtain trigrams, register dependency pairs must be combined. This could be done by combining all possible bigrams. This results in (12,16,20), (16,20,24) with the corresponding trigrams: [(instr_c, instr_a, instr_e), (instr_a, instr_e, instr_d)].

We could implement this by using list comprehensions in `python`. While this approach looks straightforward for this small example, it becomes impractical for larger functions inside large program binaries, especially when bigrams link to each other, resulting in loops.

Instead of enumerating all the possible dependency chains, we opt for another approach. We

model the register dependencies as a directed graph: every node corresponds to one instruction address, every edge corresponds to a dependency.

Now, one graph G is generated for the entire program binary. When the N -grams of a specific function need to be computed, we take the subgraph S that only contains nodes inside the specific function. The N -grams are now generated by computing all the possible walks of length N in the subgraph. This procedure is illustrated in Figure 6.3. We implement this procedure using the `NetworkX` graph library in `python`. We achieve a speedup of 1000x compared to an initial list comprehension approach. Computing trigrams for a given function now takes around $150 \mu s$ instead of 150 ms. This allows us to devise all N -grams of our dataset in near-instant time. On top of this, the approach works for larger N -grams if this would be desirable.

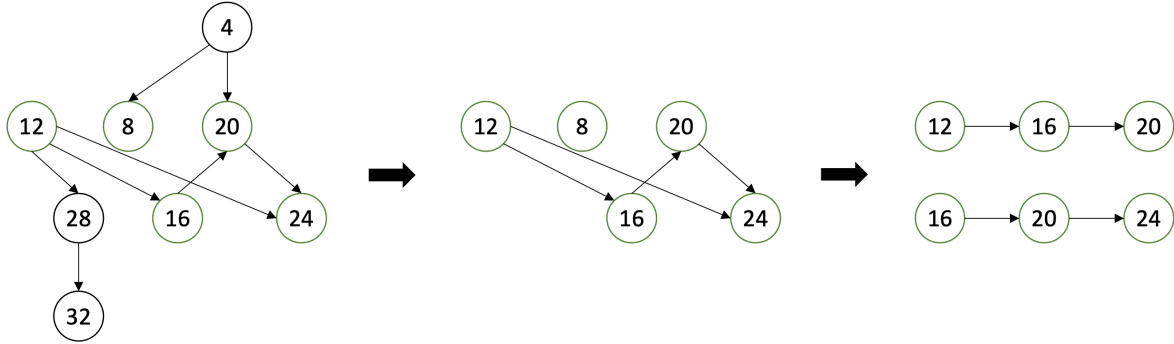


Figure 6.3: Illustration of how N -grams of a specific function are devised (here for $N=3$). First the entire program binary dependencies are mapped to a graph. Then the subgraph is taken that only contains the instructions of the function. Lastly, all the paths of length N are considered.

6.2.4 Harmonizing features in a framework

For every function, we now know all the instructions, bigrams and trigrams. These can now be converted into counts and frequencies to be fed to a machine learning model. To facilitate the workflow, we implement these transformations to be compatible with the `Scikit Learn` python framework [68]. This gives us easy access to an entire suite of machine learning tools, allowing us to use our data in conjunction with these well-established implementations.

6.3 Feature Analysis

Using the described approach, we derive the instructions, the bigrams and the trigrams for the SYNTIA dataset, the QSYNTH dataset and the ALGO dataset. We can now start with the analysis

of the features. In parallel to the natural language processing field, we will refer to instructions, digrams and trigrams as *tokens*. A *token* is the building block on which we extract the features, like frequency. Depending on which features we use, *token* refers to either instructions, bigrams or trigrams. In the following subsections, we investigate specific aspects of our datasets by using various tables and visualisations.

6.3.1 Instruction amount

In the motivating example, we argue that a window size of 10 to 50 instructions is enough to encapsulate an MBA-obfuscated segment. To assess this claim, we look at the instruction amounts per function for all three datasets. The results are summarized in figure 6.4. As expected, we see that the number of instructions increases when MBA obfuscations are applied. We also see that our assumption about the window size is wrong. Figure 6.4a indicates that SYNTIA has a minimum window size of 3 and a maximum window size of 26 for obfuscated functions. Figure 6.4b shows that QSYNTH follows the same trend as SYNTIA, albeit with slightly larger windows as the original functions are slightly more complex. The minimum window size is 4, and the maximum window size is 72. When investigating functions with the minimum window size, we notice that these are caused by a function outlining compiler optimisation. When we manually remove this optimisation, the minimum size becomes 5 for both datasets.

The distribution of the ALGO dataset is completely different from the two other datasets. This is to be expected, as the ALGO dataset contains a broader range of functions with different purposes. The assumption that one function equals an obfuscated expression is also no longer valid: most functions contain more than only mixed boolean arithmetic expressions. As a result, our labels are noisier, as we already theorised in Section 6.1.3. Nonetheless, two outliers stick out from the rest of the distribution, at an instruction amount of 975 and 903. We investigate these functions to uncover the cause. It turns out that both obfuscations are from the same program. Both functions contain some MBA expressions mixed between string assignments. More specifically, string values are changed by changing each character individually. We decide to drop these functions as they are not representative of MBA segments.

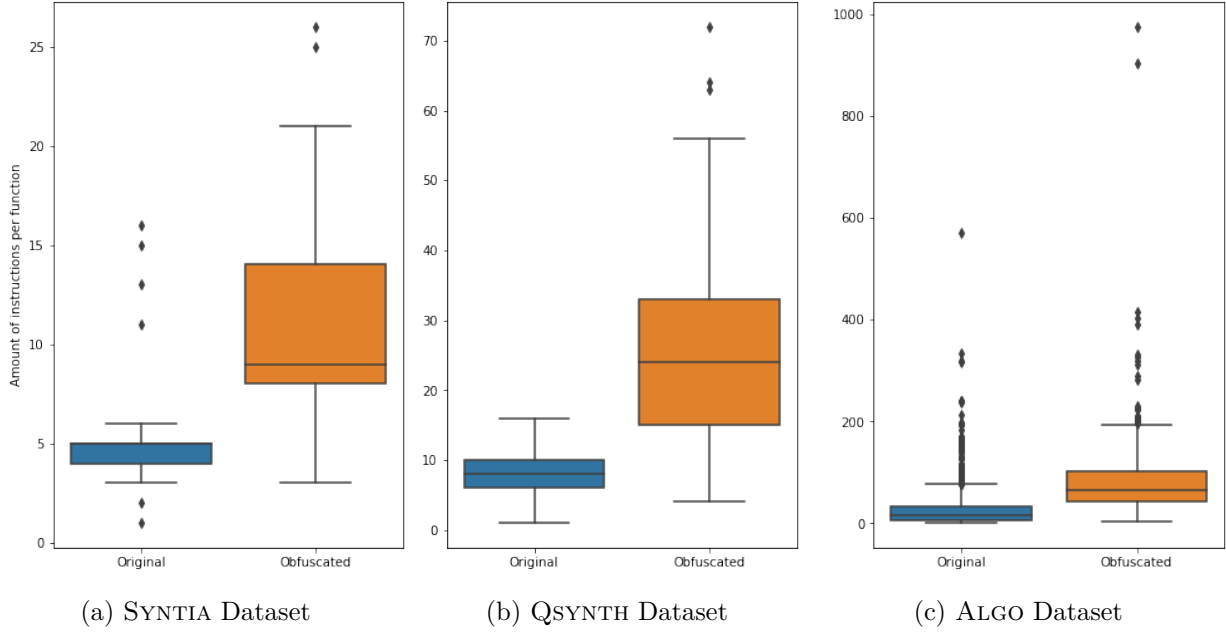


Figure 6.4: Boxplots of the amount of instructions per function for each dataset. Please note that the y-axis is not on the same scale for each plot.

In conclusion, the window sizes for obfuscated segments are outside of our assumed range. However, we do notice that most of the functions lie within the range of our assumption. Nevertheless, we want to be able to capture all sizes of MBA obfuscations and will thus increase our windows from 5 to 100 for the evaluation of unknown program binaries. We do not increase this further, as we attribute the larger window sizes of the ALGO dataset to the “impurities” of the functions.

6.3.2 Token frequency

To get an idea of the frequency of every token, we plot the distribution of these tokens for the total dataset. The occurrence of every distinct instruction is counted and divided by the total amount of instructions in the dataset. This is done for the MBA functions and the original functions separately. The same thing is done for the bigrams and the trigrams. The results are shown in Figure 6.5.

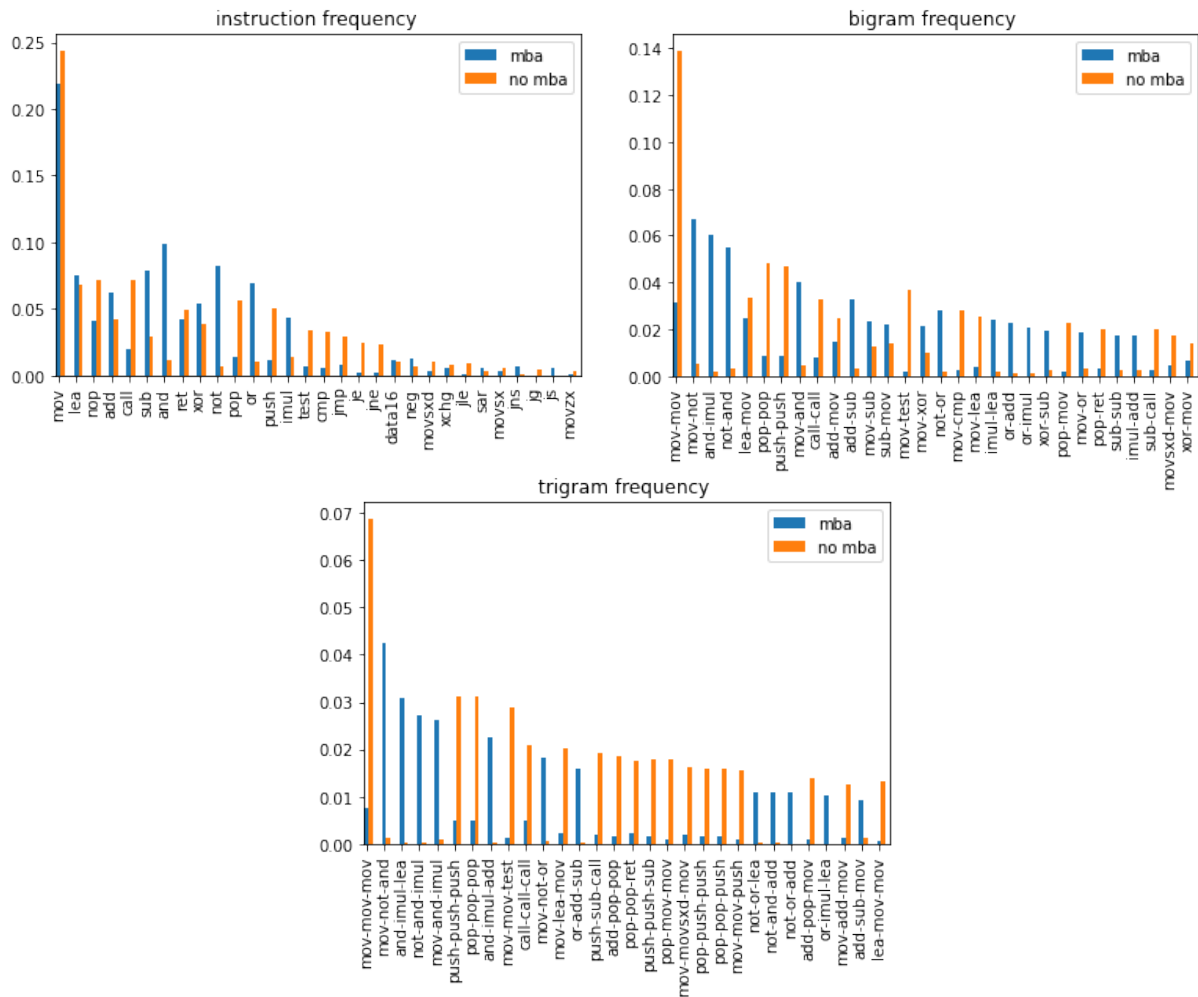


Figure 6.5: Bar plots of the total frequency of every token compared to the amount of tokens in the dataset. Please note that the y-axis is not on the same scale for each plot.

Right away, the `mov` instruction stands out from the others. It dominates all three categories. The reason for the gap between the MBA and the original set, is because the MBA sets contain more bigrams and trigrams in general, diminishing the frequency of the `mov-mov` dependency chains. By looking at the data, we conclude that the `mov` *N-grams* occur due to the temporary storage of register values in memory. When an instruction uses a value that is copied by a `mov`, it is dependent on this `mov` instruction. While this is technically correct, we are interested in the dependency between the instruction that last wrote to the `mov` source and the instruction after the `mov`.

To eliminate these `mov` dependencies, we adapt our register dependencies *pintool* to model *shortcut dependencies* instead of *raw dependencies*. *Shortcut dependencies* do not consider the intermediary `mov` instructions and show the dependency between the prior instruction and the latter

instruction instead. Figure 6.6 showcases the total total token frequencies after applying this change.

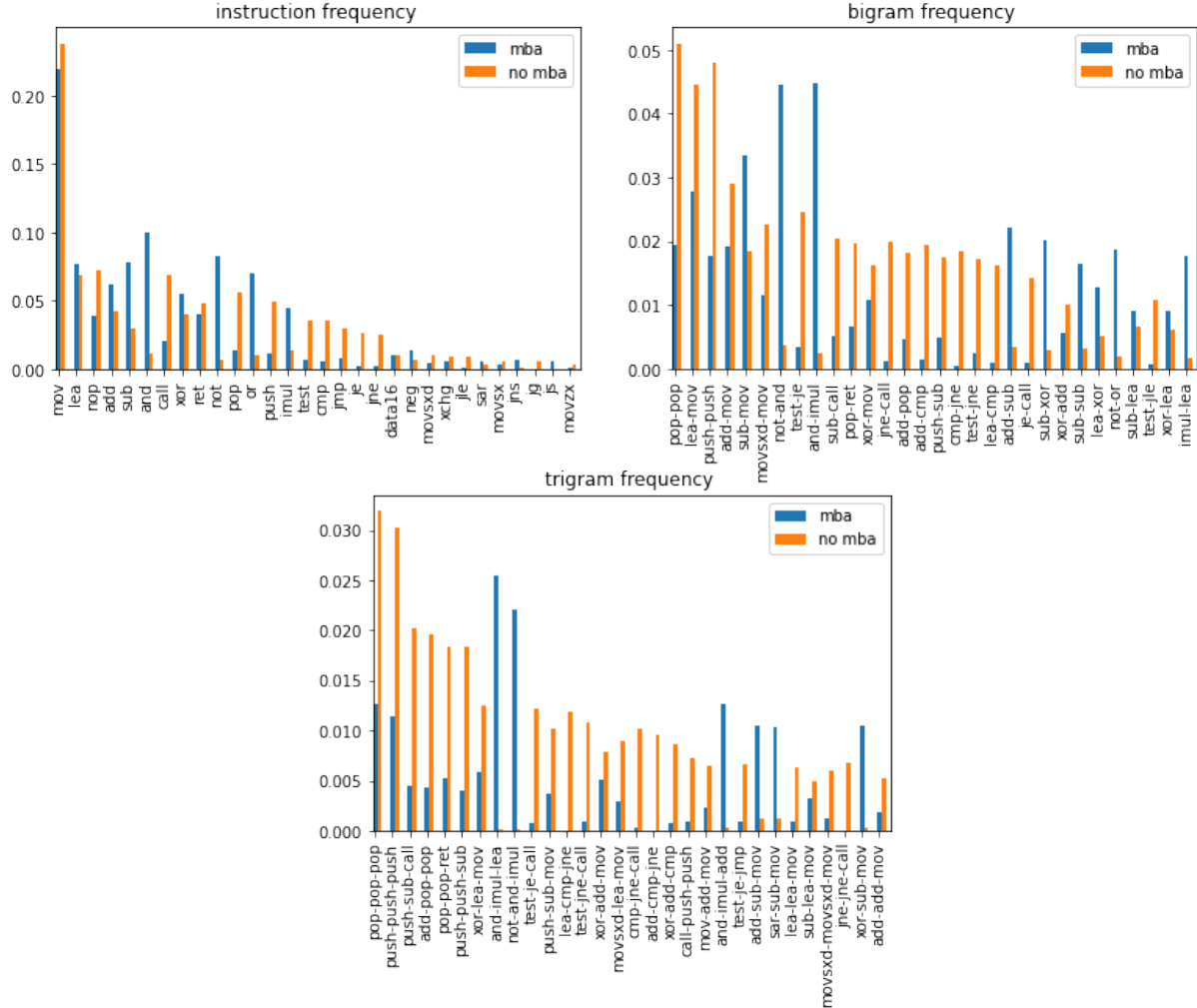


Figure 6.6: Bar plots of the total frequency of every token compared to the amount of tokens in the dataset, with shortcuts instead of raw data dependencies.

Enabling shortcuts uncovers a lot of **pop-pop** and **push-push** dependencies. When taking a closer look at the dependencies, almost all of them turn out to be caused by **rsp** register. This is a natural effect of the **push** and **pop** instructions. We decide to remove these chains as they are not indicative of MBA or non-MBA segments. After this transformation, we are left with the frequencies as shown in Figure 6.7. We can clearly distinguish feature differences between MBA and unobfuscated segments. As an example, **and-sub** and **and-imul** are clear indications of MBA obfuscations. The differences between the frequencies indicate that our features are able to capture differences between the two types of segments.

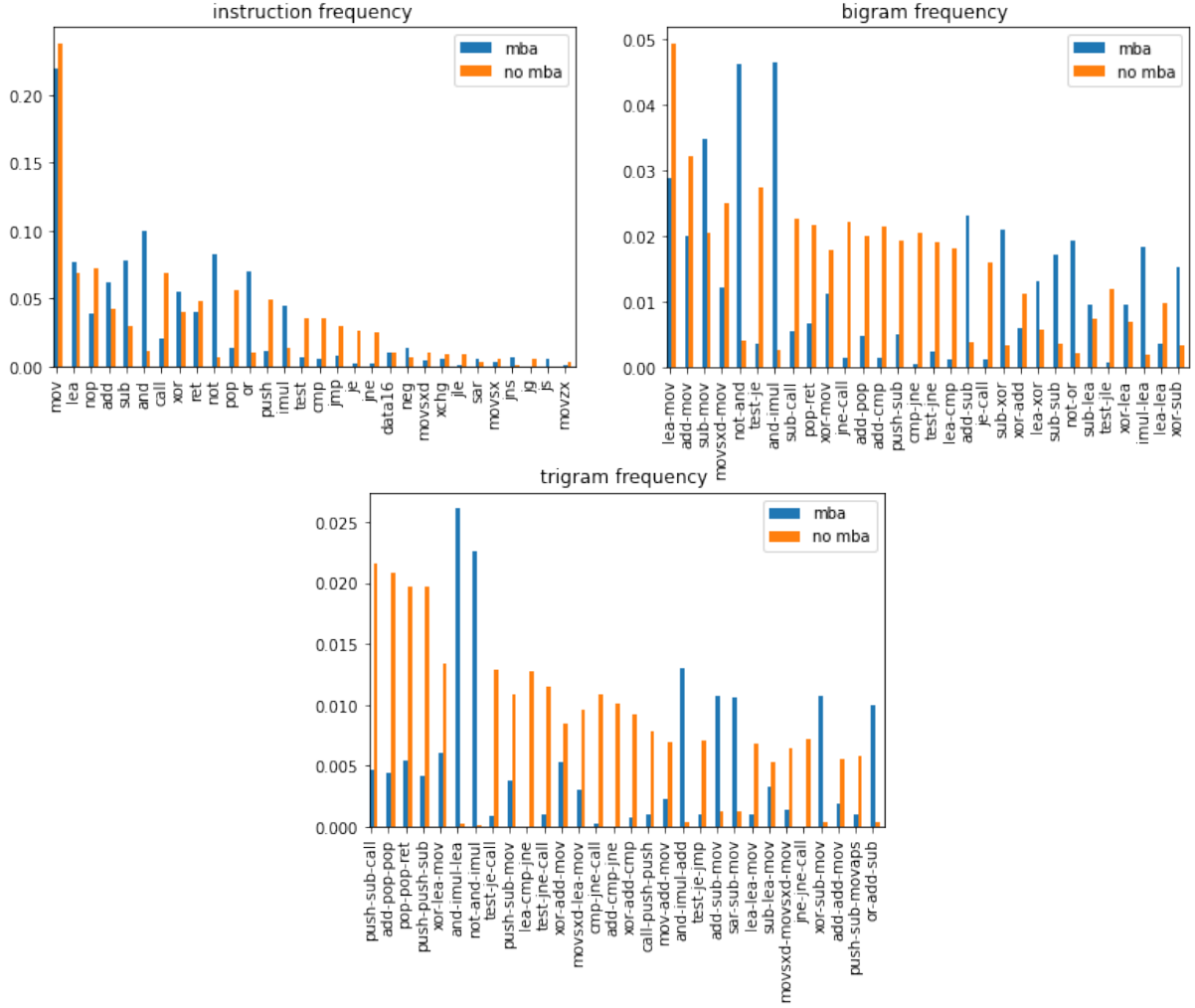


Figure 6.7: Bar plots of the total frequency of every token compared to the amount of tokens in the dataset, with shortcuts instead of raw data dependencies and without push/pop dependency chains.

6.3.3 Dataset comparison

With the resulting featureset, we are now interested in the differences between the datasets. For this, we plot the average of the token frequency per function. The resulting bar plots for each dataset are visualised in Appendix A. The following paragraphs summarise the main conclusions.

On an instruction-level (Figure A.1) the SYNTIA and QSYNTH dataset are similar. The frequency of each instruction is lower for the QSYNTH dataset, as its functions are more complex. In both cases, the average of the boolean and arithmetic instructions are different for original and MBA code. The ALGO dataset has a different frequency distribution. There are noticeably more control transfer instructions. Again, this is likely due to the ‘impureness’ of the data: the ALGO

functions contain more than just an obfuscated expression.

When looking at the bigrams (Figure A.2), we notice that the unobfuscated SYNTIA functions contain almost no bigrams. They are too short and simple. This gives a clear contrast with the obfuscated functions. On the other hand, the unobfuscated QSYNTH functions are long enough to also contain bigrams. There are noticeable differences in the frequency of the bigrams between the original and the MBA functions. However, these frequencies are not coherent with the findings of the ALGO dataset. This reaffirms the differences between the two sets. The same reasoning is applicable for the trigrams (Figure A.3).

In conclusion, the features for the SYNTIA and the QSYNTH data are quite similar. The differences are attributed to the slight increase in complexity for the QSYNTH functions. However, the ALGO dataset is vastly different. We will thus test out model performance on 1) the combination of the SYNTIA and QSYNTH dataset, 2) the ALGO dataset and 3) the entire dataset.

6.4 Model

After the feature analysis and the resulting data cleanup, we attempt to build a classifier that can distinguish between original and MBA-obfuscated segments. The final goal is to use this classifier in a *varying sliding window* approach to detect obfuscated segments in a program binary. First, the classifier will be trained and evaluated on the function datasets that are devised in Section 6.2. The evaluation of the classifier performance for any segment in a program binary is investigated in Chapter 7.

6.4.1 Train-test split

To train the classifier, the data must first be split into a train set and a test set. For this process, it is crucial to ensure that there is no data leakage between both sets. Otherwise, the model evaluations would be invalid, as they are not indicative of the performance on unseen data. To prevent data leakage, we first remove any functions with identical featuresets to prevent data samples from appearing in both sets. We also ensure that an obfuscated function is always in the same set as his original counterpart.

Now, every dataset is divided according to an 80%-20% train-test ratio. This ratio strikes a balance between keeping enough data for meaningful training and having enough test data for a representative evaluation. The resulting class distribution per set is shown in Table 6.1. The SYNTIA and QSYNTH dataset are relatively balanced. The ALGO datasets only have around 10-15% MBA functions. Not every function in the ALGO dataset contains arithmetic that can

be obfuscated. This explains the gap between the two class distributions. This class imbalance needs to be kept in mind when evaluating the performance of different models.

Dataset	Set	#MBA Functions	#Original Functions	% MBA
SYNTIA + QSYNTH	train	1170	874	57%
	test	113	92	55%
ALGO static	train	294	1607	15%
	test	73	387	16%
ALGO dynamic	train	140	1134	11%
	test	13	112	10%
Total static	train	1464	2481	37%
	test	186	479	28%
Total dynamic	train	1310	2008	39%
	test	126	204	38%

Table 6.1: Overview of the class distribution for the different datasets after removing duplicates and keeping original and obfuscated functions in the same set.

6.4.2 Evaluation Metric

Due to the imbalances in the data, we opt to use *Matthews Correlation Coefficient (MCC)* [69] as a metric instead next to *accuracy*. Accuracy alone can mislead the observer when class imbalances are present, as overpredicting the bigger class can still yield good accuracy. The MCC metric is more robust in this scenario. The MCC formula is shown in equation 6.1. MCC takes all the elements of the confusion matrix into account, compared to only the true positives and the true negatives in the case of accuracy. It yields a value between -1 and 1, where a higher score is indicative of a better model. For the evaluation of our model, we also show the *accuracy* for completeness. However, our analysis will be based on the MCC.

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (6.1)$$

$$\text{accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \quad (6.2)$$

6.4.3 Naive Bayes

We try to build a classifier by using a Multinomial Naive Bayes model. This type of model is often used in the NLP field for bag-of-words models. The classifier works by choosing the class with the maximum probability of that class given the input features:

$$y = \operatorname{argmax}_y P(y|x_1, x_2, \dots, x_n) \quad (6.3)$$

By applying Bayes' theorem and naively assuming that the features are not dependent on each other given a certain class, this equation can be rewritten as:

$$\begin{aligned} y &= \operatorname{argmax}_y P(y | x_1, \dots, x_n) \\ &= \operatorname{argmax}_y \frac{P(x_1, x_2, \dots, x_n | y)P(y)}{P(x_1, x_2, \dots, x_n)} \\ &= \operatorname{argmax}_y \frac{P(y) \prod_{i=1}^n P(x_i | y)}{P(x_1) P(x_2) \dots P(x_n)} \\ &= \operatorname{argmax}_y P(y) \prod_{i=1}^n P(x_i | y) \end{aligned} \quad (6.4)$$

The prior $P(y)$ and the posteriors $P(x_i|y)$ can now be inferred from the training data. The features for this model are the raw counts of the different tokens. To every token, a constant count is also added to prevent multiplications with posteriors of value 0. This process is called *Laplace Smoothing* [70].

The Naive Bayes model is based on the naive assumption that features are not dependent on each other, give a certain class. While this assumption does not hold for our data, this model has empirically been shown to work even when the assumption is not valid [71].

For the implementation of this model, we use our **Scikit Learn** compatible featuresets (see Section 6.2.4) together with the **Scikit Learn** Multinomial Naive Bayes Classifier. Table 6.2 and Table 6.3 summarize the results. The bag-of-instructions model is the most performant, especially for the ALGO dataset. The biggest factor is the difference in the amount of data available: the *bag-of-instructions* featureset only needs the static program binaries while the N-gram features require the dynamic instrumentation through PIN.

Dataset	Instructions	Bigrams	Trigrams
SYNTIA + QSYNTH	0.65	0.70	0.51
ALGO	0.96	0.90	0.87
All	0.69	0.49	0.42

Table 6.2: MCC scores for the multinomial naive bayes classifier using the token count as features. The rows indicate the dataset on which the model was trained and tested.

Dataset	Instructions	Bigrams	Trigrams
SYNTIA + QSYNTH	0.82	0.85	0.75
ALGO	0.98	0.97	0.97
All	0.84	0.75	0.75

Table 6.3: Accuracy scores for the multinomial naive bayes classifier using the token count as features. The rows indicate the dataset on which the model was trained and tested.

The performance of the model trained on all the data is considerably worse than the model trained on the individual datasets. This reaffirms the differences between the type of data we are dealing with, as described before.

The reason for the bad performance when working with trigrams is likely caused by the sparseness of the featureset. One obfuscated function often only contains a few trigrams, especially in the case of the SYNTIA and QSYNTH dataset. The ALGO dataset does not suffer as hard because the functions are bigger and contain more trigrams. When combining these two datasets, the small class probabilities of the SYNTIA+QSYNTH dataset are in stark contrast with the bigger class probabilities of the ALGO dataset, resulting in even worse performance when combined. However, in the case of instruction features, the combination of the two sets should result in a model that generalizes better on unseen data due to the added variance.

7

Evaluation

After the data processing and the creation and training of the classifier, we are interested in its performance for the problem at hand: enhancing the program synthesis workflow by only synthesising the segments flagged by the classifier. The method to assess this performance is explained in Section 7.1. The result of our work is then analysed in Section 7.2. Subsequently, we compare our approach to the existing approach in Section 7.3.

7.1 Evaluation Method

In Section 5.2, we set forward two criteria for a better automatic synthesis approach. To test these requirements, we first create an evaluation set of program binaries. Within these binaries, we manually label every segment that contains an MBA-obfuscated expression. We then generate every possible segment and look at the outcome of the classifier. Finally, flagged segments can be fed to a program synthesiser, like SYNTIA.

7.1.1 Evaluation Set

The constructed evaluation set consists of the following 30 program binaries:

1. Five program binaries that contain an obfuscated function similar to those found in the SYNTIA or QSYNTH dataset.
2. Five program binaries that contain an obfuscated function similar to those found in the SYNTIA or QSYNTH dataset, but compiled in a way that they are inlined in the main function.
3. Five program binaries from the ALGO dataset that each contain an obfuscated function.
4. Fifteen program binaries, consisting of the unobfuscated counterparts of the fifteen program binaries listed above.

We ensure that this evaluation set is not present in the training datasets used in Chapter 6. Otherwise, we would be dealing with data leakage and have nonmeaningful results.

For every program binary, we manually label the segments that correspond to an MBA-obfuscated expression with the help of debug information from the compilation. Contrarily to the datasets of Chapter 6, these segments do not necessarily correspond to entire functions. One function can contain multiple MBA-obfuscated expressions. Those need to be separately located in order to be simplified by program synthesis.

7.1.2 Segment labeling

To test the classifier’s performance, we generate all the possible segments of the evaluation program binaries. Just like before, this is done with a *varying sliding window* approach, now with a minimum window size of 5 and a maximum window size of 100. We categorise every segment depending on what kind of instructions they contain:

1. Segments that correspond exactly to an MBA expression.
2. Segments that only contain part of an MBA expression.
3. Segments that are bigger than, but completely contain an MBA expression.
4. Segments that contain part of an MBA expression, along with some unobfuscated code.
5. Segments that contain no MBA expressions.

Figure 7.1 clarifies the type of segments mentioned above.

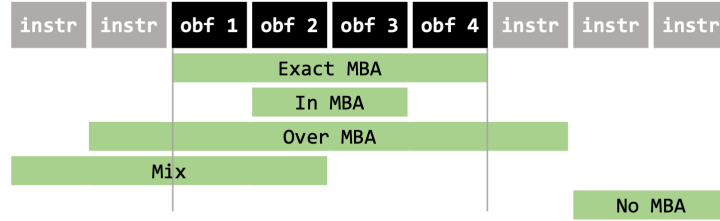


Figure 7.1: Visualisation of the different type of segments. The MBA-obfuscated segment starts at obf1 and ends at obf4. The green blocks show one segment.

Table 7.1 shows the amount of each category for the evaluation dataset. The inline dataset contains five inline MBA expressions, but the separate functions are also present in the program binary. When executing these binaries dynamically, these separate functions will never be called. We keep this in mind when evaluating our model with dynamic features. For the bigrams and the trigrams, we skip these segments.

Noticeably, one obfuscated ALGO functions contained, on average, seven MBA expressions. This confirms our hypothesis about the functions of the ALGO dataset in Section 6.1.3. We can now label every segment using our trained Naive Bayes classifier and look at the results for every category. This is done in Section 7.2

dataset	Exact MBA	In MBA	Over MBA	Mix	No MBA	Total
SYNTIA+QSYNTH	5	272	12649	7870	65114	85910
SYNTIA+QSYNTH Inline	5 (+5)	348	11111	8310	74584	94358
ALGO	31	1012	14763	7883	117245	140934
Total	41	1632	38523	24063	256943	321202

Table 7.1: Amount of segments for each category, split by the different type of program binaries in the evaluation set.

7.1.3 Program Synthesis

The final step would be to try and synthesize the segments that are labelled as MBA by the classifier. In our motivating example, we did this by using SYNTIA. However, SYNTIA is only a prototype implementation. During our tests, we ran into many issues like failed random sampling and unexpected program failures. Because of these complications, we can not derive meaningful results. Fortunately, T. Blazytko and M. Schloegel released an updated framework

at the end of July 2021, called MSYNTH [72]. However, due to the recency of the framework, we did not have the time to integrate it into our research.

Instead of performing the synthesis, we use the results from our motivating example to estimate the time it would take to synthesise the segments. We will further assume that the synthesis of MBA-obfuscated expressions is always successful when the segment is exactly one MBA-obfuscated expression. The actual results of the program synthesis are related to the program synthesiser and thus outside of the scope of this thesis.

7.2 Performance

To attest the performance of our classifier for the localisations of segments, we look at the percentage of MBA predictions for each segment category, outlined in Section 7.1.

The tables in this section display the performance of the Multinomial Naive Bayes Classifier on the Evaluation set. The rows show the segment categories. The columns show the different parts of the evaluation set. Every table corresponds to one dataset on which the classifier was trained and the type of features used. The percentage indicates the number of segments that were classified as MBA expressions.

The values should be interpreted as follows: for the *exact MBA* category, we want the percentage to be as high as possible in order to correctly label as many MBA expressions as possible. The *no MBA* values should be as low as possible: these segments offer nothing interesting when synthesised. The *mix* values are preferably also low; while they can indicate to the reverse engineer that an MBA segment is nearby, the synthesis result will not be meaningful. The same reasoning holds for the *In MBA* category. However, this category will be highly correlated with the *Exact MBA* category. The results of the *Over MBA* category are hardest to understand. Some of these segments will still produce a meaningful synthesis result (e.g. when they contain one more instruction than an exact MBA segment), but most of them will not reveal the semantics of the obfuscated segments.

7.2.1 Syntia and Qsynth dataset

Table 7.2, Table 7.3 and Table 7.4 show the evaluation results of the classifier trained on the SYNTIA and QSYNTH dataset with instructions, bigrams and trigrams respectively. The bigram and trigram versions heavily overpredict the MBA class. This is likely due to the sparseness of these features for this dataset. The unobfuscated segments contain almost no bigrams, so whenever a bigram occurs, it is almost instantly assumed to be MBA. The instruction featureset

does not suffer from this problem.

For the instruction features (Table 7.2), the results are good for the SYNTIA+QSYNTH part of the data. For the inline functions, the performance is close to the performance of the normal dataset. The prediction of inline segments is more arduous because inline segments can also contain instructions of other parts of the main that are present because of restructuring by the compiler. These instructions do not interfere with the semantics of the segment, but they do complicate the detection.

For the ALGO dataset, the performance is much lower. This is to be expected due to the different nature of the dataset. Nevertheless, a sizeable amount of MBA expressions are still identified. This is likely due to our prior hypothesis that the “clean” functions of the QSYNTH and SYNTIA data are meaningful for identifying MBA expressions in real-life functions.

dataset	Exact MBA	In MBA	Over MBA	Mix	No MBA	Total
SYNTIA+QSYNTH	100 %	98 %	0 %	2 %	0 %	1 %
SYNTIA+QSYNTH Inline	90 %	21 %	0 %	2 %	1 %	1 %
ALGO	45 %	56 %	11 %	12 %	1 %	3 %
Total	61 %	34 %	4 %	4 %	1 %	2 %

Table 7.2: Evaluation results of Naive Bayes classifier trained on the SYNTIA and QSYNTH dataset, using instructions as features.

dataset	Exact MBA	In MBA	Over MBA	Mix	No MBA	Total
SYNTIA+QSYNTH	100 %	99 %	100 %	95 %	87 %	89 %
SYNTIA+QSYNTH Inline	100 %	100 %	100 %	97 %	90 %	91 %
ALGO	81 %	84 %	73 %	74 %	97 %	93 %
Total	85 %	90 %	89 %	88 %	92 %	92 %

Table 7.3: Evaluation results of Naive Bayes classifier trained on the SYNTIA and QSYNTH dataset, using bigrams as features.

dataset	Exact MBA	In MBA	Over MBA	Mix	No MBA	Total
SYNTIA+QSYNTH	80 %	91 %	81 %	91 %	91 %	89 %
SYNTIA+QSYNTH Inline	100 %	98 %	100 %	94 %	96 %	96 %
ALGO	77 %	85 %	75 %	78 %	98 %	94 %
Total	80 %	89 %	84 %	88 %	95 %	93 %

Table 7.4: Evaluation results of Naive Bayes classifier trained on the SYNTIA and QSYNTH dataset, using trigrams as features.

7.2.2 Algo dataset

Table 7.5, Table 7.6 and Table 7.7 show the evaluation results of the classifier trained on the ALGO dataset with instructions, bigrams and trigrams respectively. Again, the model with instructions as features performs the best. In this case, this is due to the larger amount of training data for this problem. The other two models have far less training data due to their need for dynamic execution. Even with this small dataset, the bigram and trigram models are already better because the ALGO dataset suffers less from the sparseness issue: the data samples contain more trigram and bigrams in general.

dataset	Exact MBA	In MBA	Over MBA	Mix	No MBA	Total
SYNTIA+QSYNTH	100 %	97 %	27 %	27 %	0 %	7 %
SYNTIA+QSYNTH Inline	90 %	23 %	50 %	39 %	9 %	24 %
ALGO	90 %	92 %	89 %	56 %	1 %	14 %
Total	91 %	43 %	56 %	40 %	3 %	15 %

Table 7.5: Evaluation results of Naive Bayes classifier trained on the ALGO dataset, using instructions as features.

dataset	Exact MBA	In MBA	Over MBA	Mix	No MBA	Total
SYNTIA+QSYNTH	100 %	95 %	98 %	66 %	18 %	34 %
SYNTIA+QSYNTH Inline	100 %	84 %	100 %	68 %	5 %	22 %
ALGO	74 %	84 %	89 %	65 %	1 %	15 %
Total	80 %	86 %	95 %	66 %	7 %	22 %

Table 7.6: Evaluation results of Naive Bayes classifier trained on the ALGO dataset, using bigrams as features.

dataset	Exact MBA	In MBA	Over MBA	Mix	No MBA	Total
SYNTIA+QSYNTH	40 %	56 %	39 %	29 %	5 %	12 %
SYNTIA+QSYNTH Inline	60 %	56 %	56 %	41 %	0 %	10 %
ALGO	74 %	59 %	85 %	55 %	1 %	14 %
Total	68 %	58 %	62 %	42 %	2 %	12 %

Table 7.7: Evaluation results of Naive Bayes classifier trained on the ALGO dataset, using trigrams as features.

7.2.3 Entire dataset

Table 7.8, Table 7.9 and Table 7.10 show the evaluation results of the classifier trained on the entire dataset with instructions, bigrams and trigrams respectively. All three models manage to almost completely exclude *No MBA* segments. Their performance for *Exact MBA* is between 80% and 89%. It would manage to synthesise almost all MBA expressions. The trigram models and bigram models are considerably better than models trained on only certain parts of the dataset. This shows that a combination of these datasets works quite well despite their differences.

dataset	Exact MBA	In MBA	Over MBA	Mix	No MBA	Total
SYNTIA+QSYNTH	100 %	97 %	8 %	14 %	0 %	3 %
SYNTIA+QSYNTH Inline	90 %	21 %	21 %	22 %	2 %	10 %
ALGO	87 %	86 %	83 %	47 %	1 %	12 %
Total	89 %	41 %	36 %	26 %	1 %	9 %

Table 7.8: Evaluation results of Naive Bayes classifier trained on all datasets, using instructions as features.

dataset	Exact MBA	In MBA	Over MBA	Mix	No MBA	Total
SYNTIA+QSYNTH	100 %	97 %	96 %	67 %	16 %	33 %
SYNTIA+QSYNTH Inline	100 %	92 %	100 %	69 %	0 %	18 %
ALGO	74 %	86 %	89 %	64 %	1 %	15 %
Total	80 %	89 %	94 %	66 %	5 %	21 %

Table 7.9: Evaluation results of Naive Bayes classifier trained on all datasets, using bigrams as features.

dataset	Exact MBA	In MBA	Over MBA	Mix	No MBA	Total
SYNTIA+QSYNTH	100 %	81 %	95 %	49 %	5 %	23 %
SYNTIA+QSYNTH Inline	100 %	68 %	100 %	61 %	0 %	17 %
ALGO	74 %	62 %	87 %	57 %	2 %	14 %
Total	80 %	66 %	93 %	56 %	2 %	17 %

Table 7.10: Evaluation results of Naive Bayes classifier trained on all datasets, using trigrams as features.

7.3 Comparison with initial approach

In this section, we investigate if our approach is better than the initial workflow described in Section 5.1. To this end, we set forward two requirements for an improved design:

1. Limit the number of times that SYNTIA is invoked in order to cut down on computation time and noise due to uninteresting results.
2. Be accurate enough in the identification of MBA windows in order to be able to recover the wanted semantics.

Section 7.2 shows that the second requirement is partially met: our classifier approach is able to correctly locate between 89% of the MBA expressions, depending on the featureset. For the first requirement, we see that Syntia is invoked 9% of the time. This is considerably better than synthesising every segment. The reverse engineer also knows that a flagged segment indicates that the MBA segment is nearby, as almost all the *No MBA* segments are correctly ignored. So the noise of unmeaningful synthesis results is reduced and can still indicate the nearby presence of an MBA obfuscation.

For time efficiency, we time the different components of our approach on the evaluation set. The time to generate all possible components, together with the register dependencies, took 80 seconds per program binary on average. The time to generate the different bigrams and trigrams from the 30 program binaries (321202 segments) takes 4 minutes in total. The classification of all those segments takes 30 seconds. For one program binary, the resulting average time is 90 seconds. This time is negligible compared to the program synthesis time. As the amount of program synthesis executions is reduced by 91%, the overall time is roughly reduced by 90%.

8

Conclusion

In this thesis, we designed a method to automatically locate mixed boolean arithmetic inside a program binary. We set forward a classification approach that attempts to label every possible segment in a program binary, obtained with a *varying sliding window* algorithm, as either unobfuscated or MBA-obfuscated.

We started by examining existing data sources, like the datasets of the SYNTIA [59] and QSYNTH [64] framework, and concluding with the need for an additional dataset. We devised this set of program binaries based on a collection of C programs called THE ALGORITHMS [67]. Instead of manually labelling every MBA-obfuscated segment inside each binary, we opted for an automatic approach that labels entire functions as obfuscated or unobfuscated.

From the three mentioned datasets, we derived three types of features: a *bag-of-instructions*, *register dependency bigrams* and *register dependency trigrams*. To obtain these features, the instructions were derived from an `objdump` assembly reconstruction of the program binaries. We used a custom *pintool* to derive the register dependencies for each binary dynamically. These results were then translated into a graph construction to efficiently retrieve the bigrams and trigrams.

With the constructed features, we trained a Multinomial Naive Bayes classifier to differentiate between MBA-obfuscated functions and unobfuscated functions. We then evaluated the classi-

fiers ability to label any segment in a program binary as either an MBA-obfuscated expression or non-obfuscated code.

Our classifier can identify 89% of the MBA-obfuscated segments when using instructions as features and 80% of the MBA-obfuscated segments when using register dependency bigrams and trigrams. It also accurately identifies segments that contain no MBA expressions. For segments that contain part of an MBA expression, the classifier labels around 30% as MBA. While this is not necessarily desirable, these segments indicate to the reverse engineer that an MBA expression is somewhere in this area of the program binary.

Future Work

Our approach is based on supervised machine learning classification. In the scope of this thesis, a classifier was used to locate MBA-obfuscated segments. However, with the procured data and constructed features, this classifier can be extended to the classification and locating of other types of obfuscation. The featureset can also be expanded by taking a look at the memory dependencies between instructions.

For this particular research, a next step could be the omission of the *sliding window* component. Instead, the following segmentation approach can be researched: train a classifier to classify the first and last instruction of an MBA-obfuscated expression. In this way, the MBA-obfuscated segment can be located in the program binary. The bigrams and trigrams can serve as features for each instruction. These dependency chains give a sense of previous and future context to each instruction.

In general, machine learning processes requires meaningful data in order to be trained and evaluated correctly. This is a bottleneck for the adaptation of machine learning in the field of reverse engineering. This field could benefit from more data sources to drive this kind of research forward.

Bibliography

- [1] N. Eyrolles, “Obfuscation with mixed boolean-arithmetic expressions: reconstruction, analysis and simplification tools,” Ph.D. dissertation, Université Paris-Saclay, 2017.
- [2] T. László and Á. Kiss, “Obfuscating c++ programs via control flow flattening,” *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, vol. 30, no. 1, pp. 3–19, 2009.
- [3] R. David, L. Coniglio, and M. Ceccato, “Qsynth-a program synthesis based approach for binary code deobfuscation,” in *BAR 2020 Workshop*, 2020.
- [4] J. Lazenby, *The First Punic War*. Routledge, 2016.
- [5] E. J. Chikofsky and J. H. Cross, “Reverse engineering and design recovery: A taxonomy,” *IEEE software*, vol. 7, no. 1, pp. 13–17, 1990.
- [6] M. L. Nelson, “A survey of reverse engineering and program comprehension,” *arXiv preprint cs/0503068*, 2005.
- [7] P. Hall, “Overview of reverse engineering and reuse research,” *Information and Software Technology*, vol. 34, no. 4, pp. 239 – 249, 1992. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/0950584992900809>
- [8] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski, “Guest editors’ introduction: Software protection,” *IEEE Software*, vol. 28, no. 2, pp. 24–27, 2011.
- [9] Y. Gu, B. Wyseur, and B. Preneel, “Software-based protection is moving to the mainstream,” *IEEE Software, Special Issue on Software Protection*, vol. 28, no. 2, pp. 56–59, 2011.
- [10] O. Shy and J.-F. Thisse, “A strategic approach to software protection,” *Journal of Economics & Management Strategy*, vol. 8, no. 2, pp. 163–190, 1999.
- [11] J. Nagra and C. Collberg, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

- [12] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, “On the (im) possibility of obfuscating programs,” *Journal of the ACM (JACM)*, vol. 59, no. 2, pp. 1–48, 2012.
- [13] S. A. Sebastian, S. Malgaonkar, P. Shah, M. Kapoor, and T. Parekhji, “A study & review on code obfuscation,” in *2016 World Conference on Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave)*. IEEE, 2016, pp. 1–6.
- [14] A. Balakrishnan and C. Schulze, “Code obfuscation literature survey,” *CS701 Construction of compilers*, vol. 19, 2005.
- [15] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” Citeseer, Tech. Rep., 1997.
- [16] Hex-rays, “Ida - interactive disassembler,” <https://hex-rays.com/ida-pro/>.
- [17] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, “Static disassembly of obfuscated binaries,” in *USENIX security Symposium*, vol. 13, 2004, pp. 18–18.
- [18] GNU Project, “Objdump - display information from object files,” <https://www.gnu.org/software/binutils/>.
- [19] D. S. Katz, J. Ruchti, and E. Schulte, “Using recurrent neural networks for decompilation,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 346–356.
- [20] GNU Project, “Gdb: The gnu project debugger,” <https://www.gnu.org/software/gdb/>.
- [21] Intel, “Pin - a dynamic binary instrumentation tool,” <https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html>.
- [22] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.
- [23] C. Barrett, D. Kroening, and T. Melham, *Problem solving for the 21st century: Efficient solver for satisfiability modulo theories*, ser. Knowledge Transfer Report, Technical Report 3. London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering, Jun. 2014.
- [24] J. Salwan and F. Soudel, “Triton: Framework d’exécution concolique et d’analyses en runtime,” in *Proceedings of the 2015 Symposium sur la Securite des Technologies de l’Information et des Communications, ser. SSTIC*, vol. 15, 2015.
- [25] K. Sen, “Concolic testing,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 571–572.

- [26] CEA IT Security, “Miasm - reverse engineering framework in python,” <https://miasm.re>.
- [27] Quarkslab, “Triton - dynamic binary analysis,” <https://github.com/JonathanSalwan/Triton>.
- [28] M. Leotta, M. Biagiola, F. Ricca, M. Ceccato, and P. Tonella, “A family of experiments to assess the impact of page object pattern in web test suite development,” in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 263–273.
- [29] University of Arizona, “The tigress c obfuscator,” <http://tigress.wtf>.
- [30] Orleans Technologies, “Themida - advanced windows software protection system,” <http://oreans.com/themida.php>.
- [31] S. Berlato and M. Ceccato, “A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps,” *Journal of Information Security and Applications*, vol. 52, p. 102463, 2020.
- [32] A. Majumdar, C. Thomborson, and S. Drape, “A survey of control-flow obfuscations,” in *International Conference on Information Systems Security*. Springer, 2006, pp. 353–356.
- [33] D. Xu, J. Ming, and D. Wu, “Generalized dynamic opaque predicates: A new control flow obfuscation method,” in *International Conference on Information Security*. Springer, 2016, pp. 323–342.
- [34] L. Zobernig, S. D. Galbraith, and G. Russello, “When are opaque predicates useful?” in *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 2019, pp. 168–175.
- [35] P. Zhao and J. N. Amaral, “Function outlining and partial inlining,” in *17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD’05)*. IEEE, 2005, pp. 101–108.
- [36] Y. Li, Z. Sha, X. Xiong, and Y. Zhao, “Code obfuscation based on inline split of control flow graph,” in *2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*. IEEE, 2021, pp. 632–638.
- [37] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, “Information hiding in software with mixed boolean-arithmetic transforms,” in *International Workshop on Information Security Applications*. Springer, 2007, pp. 61–75.
- [38] B. Liu, J. Shen, J. Ming, Q. Zheng, J. Li, and D. Xu, “Mba-blast: Unveiling and simplifying mixed boolean-arithmetic obfuscation,” in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

- [39] D. Xu, J. Ming, Y. Fu, and D. Wu, “Vmhunt: A verifiable approach to partially-virtualized binary code simplification,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 442–458.
- [40] R. Rolles, “Unpacking virtualization obfuscators,” in *3rd USENIX Workshop on Offensive Technologies.(WOOT)*, 2009.
- [41] VMProtect Software, “Vmprotect software protection,” <http://vmpsoft.com/>.
- [42] Sony DADC, “Securom software protection,” <http://securom.com/>.
- [43] S. Gulwani, “Dimensions in program synthesis,” in *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, 2010, pp. 13–24.
- [44] S. Gulwani, O. Polozov, and R. Singh, “Program synthesis,” *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017. [Online]. Available: <http://dx.doi.org/10.1561/25000000010>
- [45] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, “Synthesis of loop-free programs,” in *PLDI’11, June 4-8, 2011, San Jose, California, USA*, 2011.
- [46] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, “Syntia: Synthesizing the semantics of obfuscated code,” in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 643–659.
- [47] Unicorn Engine, “Unicorn - the ultimate cpu emulator,” <https://www.unicorn-engine.org/>.
- [48] Y. Zhou and A. Main, “Diversity via code transformations: A solution for ngna renewable security,” *NCTA-The National Show*, 2006.
- [49] C. Liem, Y. Zhou, and Y. X. Gu, “System and method providing dependency networks throughout applications for attack resistance,” Nov. 7 2017, uS Patent 9,811,666.
- [50] H. J. Johnson, Y. X. Gu, and Y. Zhou, “System and method of interlocking to protect software-mediated program and device behaviours,” Jun. 10 2014, uS Patent 8,752,032.
- [51] A. N. Kandanchatha and Y. Zhou, “System and method for obscuring bit-wise and two’s complement integer computations in software,” Jun. 21 2011, uS Patent 7,966,499.
- [52] H. S. Warren, *Hacker’s delight*. Pearson Education, 2013.
- [53] Quarkslab, “Quarks app protect,” <https://quarkslab.com/quarks-appshield-app-protect/>.
- [54] Irdeto, “Cloakware by irdeto,” <https://irdeto.com/cloakware-by-irdeto/>.

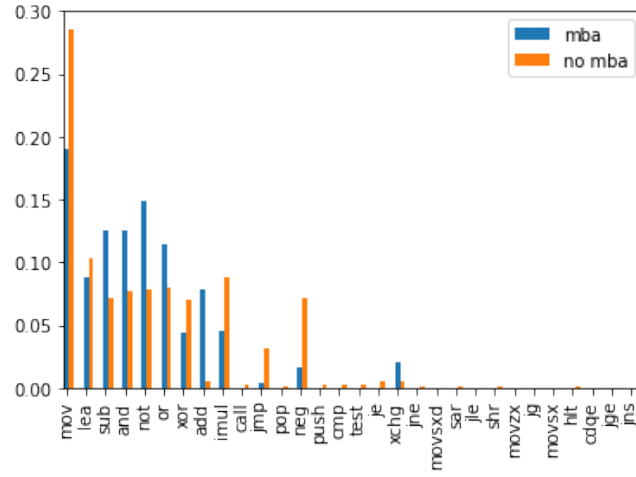
- [55] MapleSoft, “Maple - powerful math software that is easy to use,” <https://maplesoft.com/products/Maple/>.
- [56] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
- [57] A. Guinet, N. Eyrolles, and M. Videau, “Arybo: Manipulation, canonicalization and identification of mixed boolean-arithmetic symbolic expressions,” in *GreHack 2016*, 2016.
- [58] N. Eyrolles, L. Goubin, and M. Videau, “Defeating mba-based obfuscation,” in *Proceedings of the 2016 ACM Workshop on Software PROtection*, 2016, pp. 27–38.
- [59] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, “Syntia framework github repository,” <https://github.com/RUB-SysSec/syntia>.
- [60] GNU Project, “Gcc - the gnu compiler collection,” <https://gcc.gnu.org/>.
- [61] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson Education, 2016.
- [62] D. Jurafsky and J. H. Martin, *Speech and Language Processing - An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition (3rd ed. draft)*, 2020, <https://web.stanford.edu/~jurafsky/slp3>.
- [63] “Github repo of this master thesis,” https://github.ugent.be/ajdschri/Thesis_MBA.
- [64] R. David, L. Coniglio, and M. Ceccato, “Qsynth dataset github repository,” <https://github.com/werew/qsynth-artifacts>.
- [65] OpenSSL Software Foundation, “Openssl - cryptography and ssl/tls toolkit,” www.openssl.org.
- [66] University of Arizona, “The tigress c obfuscator - usage information,” <https://tigress.wtf/usage.html>.
- [67] The Algorithms, “The algorithms - github’s largest open-source algorithm library,” <https://the-algorithms.com>.
- [68] Inria Foundation, “Scikit learn python module,” <https://scikit-learn.org/>.
- [69] B. W. Matthews, “Comparison of the predicted and observed secondary structure of t4 phage lysozyme,” *Biochimica et Biophysica Acta (BBA)-Protein Structure*, vol. 405, no. 2, pp. 442–451, 1975.
- [70] D. Jurafsky and J. H. Martin, *Speech and Language Processing (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2009.

- [71] I. Rish *et al.*, “An empirical study of the naive bayes classifier,” in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3, no. 22, 2001, pp. 41–46.
- [72] T. Blazytko and M. Schloegel, “Msynth framework github repository,” <https://github.com/mrphrazer/msynth>.

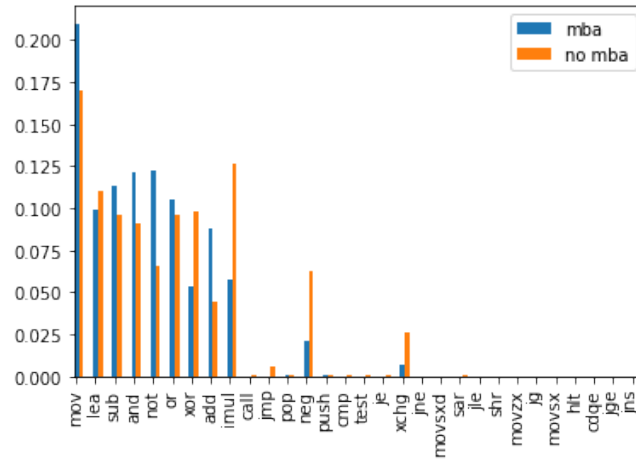
Appendices

Appendix A - Visualisation of the distribution of each dataset

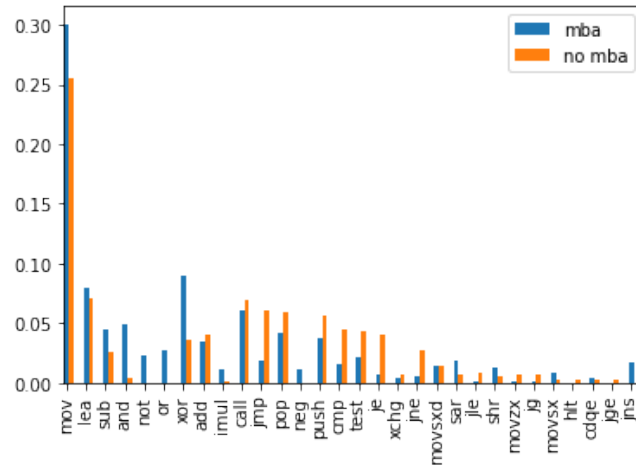
This appendix contains bar plots for the distribution of the instructions, the bigrams, and the trigrams for each dataset. For every dataset, the frequency of a token inside a function is averaged over the entire dataset. Each dataset is split into a set of unobfuscated functions, and a set of obfuscated functions. For the sake of interpretability, the tokens are ordered according to their total frequency for the entire dataset. In this way, the different bar plots can easily be compared. Please note that the scale of the y-axis is not always the same.



(a) Average instruction frequency per SYNTIA function.

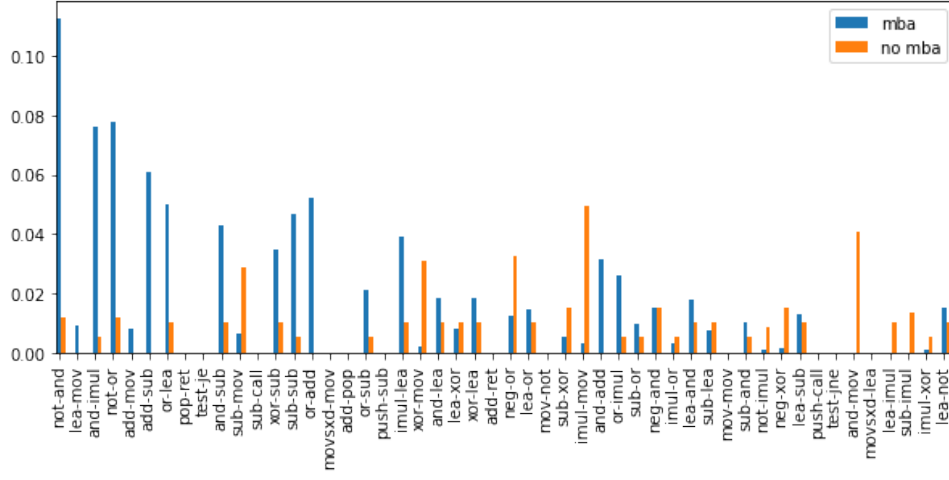


(b) Average instruction frequency per QSYNTH function.

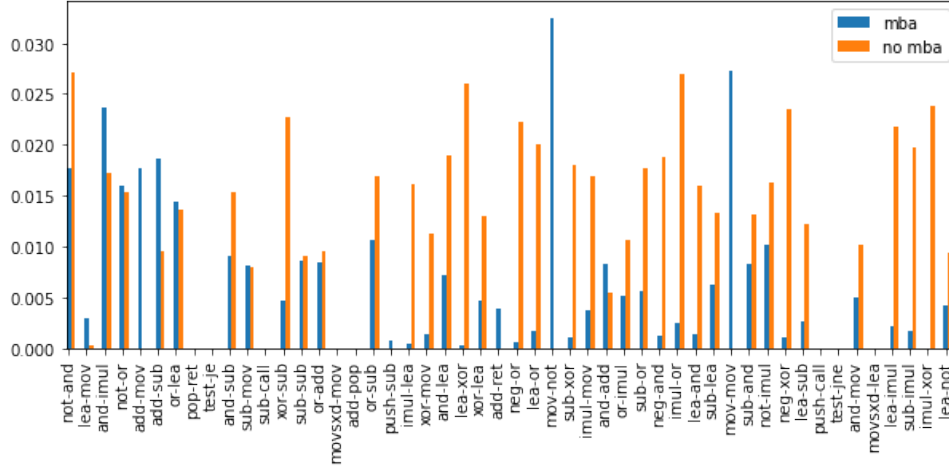


(c) Average instruction frequency per ALGO function

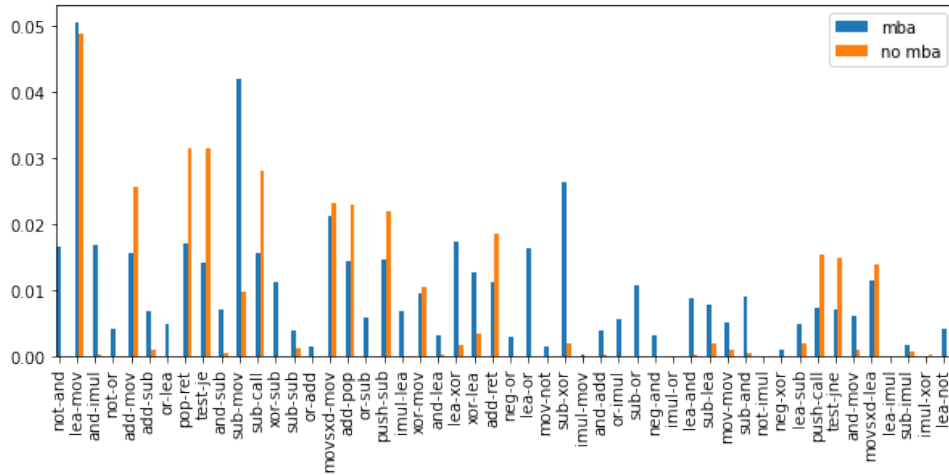
Figure 1: Bar plots of the average instruction frequency per instruction for each dataset.



(a) Average bigram frequency per SYNTIA function.

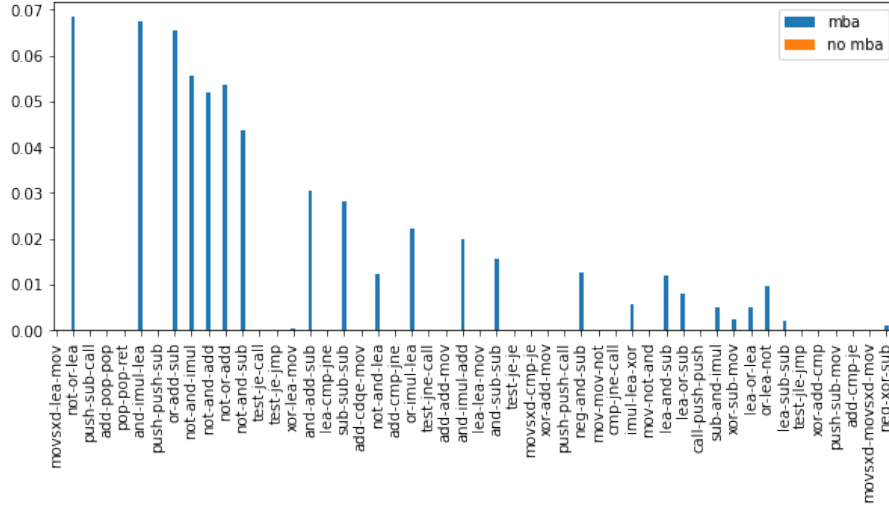


(b) Average bigram frequency per QSYNTH function.

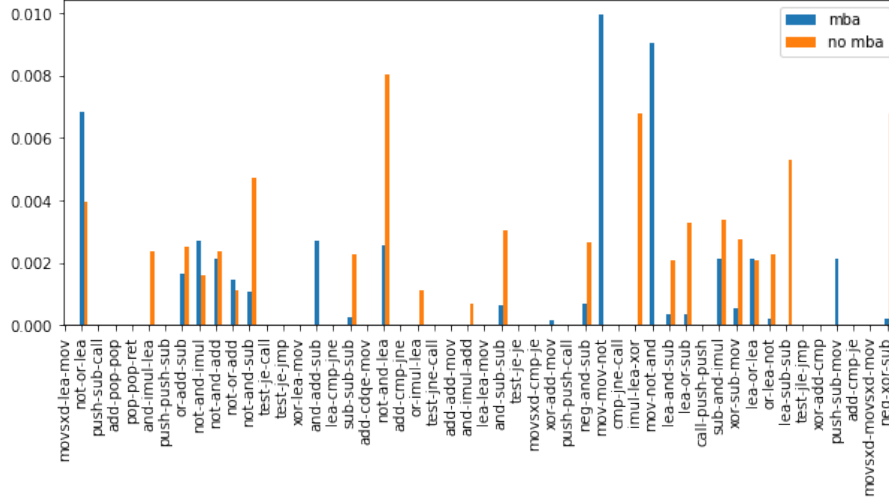


(c) Average bigram frequency per ALGO function

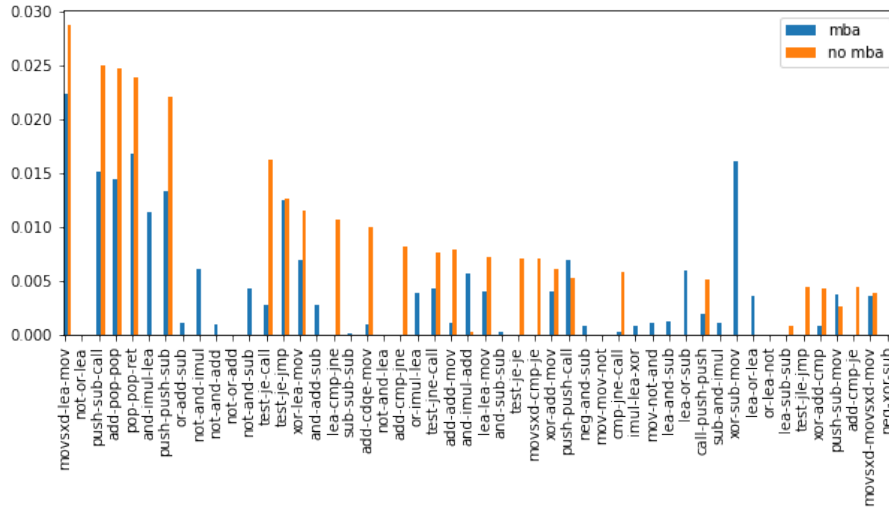
Figure 2: Bar plots of the average bigram frequency per instruction for each dataset.



(a) Average trigram frequency per SYNTIA function.



(b) Average trigram frequency per QSYNTH function.



(c) Average trigram frequency per ALGO function

Figure 3: Bar plots of the average trigram frequency per instruction for each dataset.

Automated Localisation of a Mixed Boolean Arithmetic Obfuscation Window in a Program Binary

Antoine De Schrijver

Student number: 01506917

Supervisors: Prof. dr. ir. Bjorn De Sutter, Dr. Bart Coppens

Counsellor: Dr. ir. Bert Abrath

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2020-2021