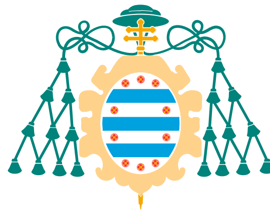
Obtaining High-Level Semantic Information from Binary Code



Javier Escalada Gómez

PhD Supervisor

Prof. Francisco Ortín Soler

Doctoral Program in Computer Science

University of Oviedo

A thesis submitted for the degree of

Doctor of Philosophy

Oviedo, Spain

April 2021

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

RESUMEN DEL CONTENIDO DE TESIS DOCTORAL

1.- Título de la Tesis	
Español: Obtención de Información Semántica de Alto Nivel a partir de Código Binario	Inglés: Obtaining High-Level Semantic Information from Binary Code

2.- Autor	
Nombre: Javier Escalada Gómez	DNI
Programa de Doctorado: Informática	
Órgano responsable: Centro Internacional de Postgrado	

FOR-MAT-VOA-010 (Reg.2018)

RESUMEN (en español)

La decompilación es el proceso dentro de la ingeniería inversa de software encargado de recuperar el código fuente de alto nivel a partir de un fichero binario. Aunque el código fuente obtenido tiene el mismo comportamiento que el programa original, su legibilidad suele ser inferior y casi siempre es distinto al código escrito por el programador. La obtención del programa original a partir de un binario es considerado como un problema indecidible puesto que, cuando se compila un programa, el compilador descarta mucha información de alto nivel no necesaria en el binario generado.

Para tratar de reconstruir esta información, los decompiladores asocian patrones de instrucciones ensamblador a construcciones de alto nivel, permitiendo obtener construcciones de alto nivel a partir de patrones binarios. No obstante, la obtención de los patrones binarios es una tarea compleja y éstos dependen de variables tales como el compilador utilizado, opciones de compilación, el lenguaje de alto nivel de origen y el procesador para el que se genera el binario, entre otros. Si alguna de estas variables cambia, los patrones binarios a detectar también cambian.

En los últimos años se están utilizando técnicas de aprendizaje automático y *big data* para construir herramientas orientadas a mejorar el desarrollo de software. La línea de investigación *big code* obtiene grandes volúmenes de código fuente a partir de repositorios de código fuente para entrenar modelos probabilísticos predictores. Con este tipo de técnicas se han construido herramientas de diversa índole, tales como desofuscadores de código, traductores automáticos o detectores de errores y vulnerabilidades en código fuente. Esta tesis plantea en qué medida este tipo de técnicas pueden ser utilizadas para resolver el problema indecidible de la decompilación.

Esta tesis describe un método que, utilizando aprendizaje automático supervisado, mejora la extracción de información de alto nivel en comparación con los decompiladores existentes en el mercado. Inicialmente, el método plantea la creación de distintos modelos en función de las variables influyentes en la detección de los patrones binarios, eligiendo uno de los modelos creados una vez se hayan detectado los valores de las variables influyentes. Otro de los retos abordados por el método es la enorme variabilidad en la representación binaria de instrucciones ensamblador. Para ello, se define proceso de generalización de patrones en el que el aprendizaje automático es utilizado junto con un experto de dominio para reducir la variabilidad de dichos patrones. Finalmente, un proceso iterativo es el encargado de ayudar al experto en la creación de modelos que puedan inferir nuevas construcciones en lenguajes de alto nivel. El método propuesto no se limita a utilizar aprendizaje automático para entrenar los modelos, sino que también lo emplea como un mecanismo de descubrimiento de características (*feature engineering*).

Para entrenar los modelos predictivos de manera automática, diseñamos e implementamos una plataforma altamente parametrizable encargada de generar los conjuntos de datos usados en el entrenamiento. Para ello, primero permite instrumentar el código fuente de alto nivel para relacionar las construcciones de alto nivel y su representación binaria. Posteriormente se extraen estos patrones binarios iniciales y se aplican las generalizaciones previamente identificadas por el método descrito en el párrafo anterior. Finalmente se compone el conjunto de datos, caracterizando cada instancia, fila o individuo mediante los patrones binarios

obtenidos y etiquetándolo con la construcción de alto nivel oportuna. La implementación ha sido altamente paralelizada y por ello es capaz de obtener una mejora de 3,5 factores sobre un máximo teórico de 4, cuando se ejecuta en un procesador con 4 núcleos.

Nuestros modelos se entrenan con código fuente C, pero es difícil obtener grandes volúmenes de código C estándar que pueda compilarse en diversos compiladores. Por ello implementamos un generador de código estocástico llamado Cnerator. Este generador, nos permite generar grandes volúmenes de código fuente con descripciones probabilísticas de las distintas construcciones del lenguaje, así como asegurar que se cumplen determinadas reglas especificadas por el usuario. El código sintético generado por Cnerator es enriquecido con código real obtenido de repositorios de código fuente. Así conseguimos cubrir un elevado espacio de búsqueda (código sintético) y representar las construcciones típicas mas utilizadas por los programadores (código real).

Con todo ello, se construyó un modelo predictivo orientado a detectar el tipo de alto nivel retornado por todas las funciones existentes en código binario. Tras construir el conjunto de datos, se crearon 14 modelos clasificadores con distintos algoritmos de aprendizaje automáticos, evaluándolos con 3 métodos que consideran su capacidad predictiva ante código no utilizado en el entrenamiento. Todos los modelos obtuvieron mejores resultados que todos los decompiladores existentes, para los 3 métodos de evaluación. Comparando la capacidad predictiva del mejor modelo y el mejor decompilador, nuestro sistema obtuvo un F1-score del 79,1 % frente al 30 % F1-score del mejor decompilador del mercado.

Tras la creación de los conjuntos de datos para entrenar los modelos, realizamos un análisis de los primeros para documentar los patrones utilizados en la clasificación del tipo de retorno de las funciones. Con este fin obtuvimos y analizamos reglas que asocian patrones binarios a los tipos de retorno de alto nivel. Dichas reglas de asociación combinan patrones binarios obtenidos dentro del cuerpo de una función con patrones de utilización del valor devuelto tras las invocaciones a la función. Las reglas combinan la información binaria de cómo una función retorna un valor con información de cómo se usa dicho valor, para detectar los tipos de retorno de alto nivel. Estos patrones pueden ser incluidos en los decompiladores actuales y así mejorar la inferencia que hacen de los tipos de retorno de las funciones decompiladas.

RESUMEN (en inglés)

In software reverse engineering, decompilation is the process of recovering source code from binary files. Decompilers are used when it is necessary to understand or analyze software for which the source code is not available. Although existing decompilers commonly obtain source code with the same behavior as the binaries, that source code is usually hard to interpret and certainly differs from the original code written by the programmer. This is because obtaining the original source code from a binary file is an undecidable problem. The cause is that the compiler discards high-level information in the translation process, such as type information, that cannot be recovered in the inverse process.

Existing decompilers associate binary patterns with high-level language constructs so that binary code can be decompiled. However, those binary patterns strongly depend on different variables such as the compiler used to generate the binaries, target microprocessor and operating system, and the compiler options. If the values of one of these variables change, so do the binary patterns.

In the last years, machine learning and big data techniques have been used to build tools aimed at improving software construction. The “big code” research line is focused on using massive code-bases from open source-code repositories to train predictive models. With this approach, different software development tools have been built, such as deobfuscators, automatic source-code translators, and error and vulnerability detectors. This dissertation poses how to use these kinds of techniques to tackle the undecidable problem of decompilation.

This dissertation proposes a method that, using supervised machine learning, improves the high-level semantic information inferred by existing decompilers. First, it identifies the variables that influence the models to be created. Different models are created, using the appropriate one depending on the values of the influencing variables. Another challenge faced by the method is the strong variability of binary code. Together with a domain expert, supervised machine learning is used to generalize the binary patterns and reduce the high dimensionality of the

problem. Moreover, another iterative process is defined to assist the expert in the creation of decompilation models for different language constructs. Therefore, machine learning is not only used to train the models, but also as a feature engineering method.

We design and implement a platform for the automatic creation of datasets to build predictive models. It facilitates the instrumentation of source code that helps to associate binary patterns and high-level language constructs. Binary patterns are extracted and generalized (as explained above) to reduce the number of features and improve the performance of the models. The datasets are then created, identifying the binary patterns for each instance (individual or row), and labeled with the target high-level construct. The platform implementation has been highly parallelized, obtaining 3.5 factors of speedup out of a theoretical maximum value of 4 factors (in a 4-core multiprocessor).

Our models are trained with C source code, but it is not easy to gather massive code-bases of standard C projects. For this reason, we implemented Cnerator, a stochastic C source-code generator. It is highly configurable, allowing the user to specify the probability distributions of each language construct, properties of the generated code, and post-processing modifications of the output programs. The synthetic code generated is added to different real open-source projects. In this way, we cover a high search space (synthetic code) and the common C idioms used by real programmers.

Using all these elements, we apply the proposed method to decompile the high-level type returned by functions. We first build the dataset with synthetic and real programs. Then, we train 14 different models using traditional machine learning algorithms. Those models are evaluated with 3 distinct methods and compared with the state-of-the-art decompilers. All the models outperform the existing decompilers, for the 3 evaluation methods. The best predictive model obtains a 79.1% F1-score, whereas the best decompiler gets a 30% F1-score.

The dataset created is also used to document the binary patterns used to predict the high-level return type. We create association rules that correlate binary patterns with C return types. Such patterns combine the binary code used to return a value (inside the function body), and the binary patterns that use the returned expression after the function is called. This information is a valuable asset to improve the implementation of existing decompilers.

Acknowledgements

This work has been partially funded by the Spanish Department of Science, Innovation and Universities: project RTI2018-099235-B-I00. It has also been funded by the University of Oviedo through its support to official research groups (GR-2011-0040).

Part of the research presented in this dissertation has been funded by the European Union, through the European Regional Development Funds (ERDF); and the Principality of Asturias, through its Science, Technology and Innovation Plan (grant GRUPIN14-100).

Abstract

In software reverse engineering, decompilation is the process of recovering source code from binary files. Decompilers are used when it is necessary to understand or analyze software for which the source code is not available. Although existing decompilers commonly obtain source code with the same behavior as the binaries, that source code is usually hard to interpret and certainly differs from the original code written by the programmer. This is because obtaining the original source code from a binary file is an undecidable problem. The cause is that the compiler discards high-level information in the translation process, such as type information, that cannot be recovered in the inverse process.

Existing decompilers associate binary patterns with high-level language constructs so that binary code can be decompiled. However, those binary patterns strongly depend on different variables such as the compiler used to generate the binaries, target microprocessor and operating system, and the compiler options. If the values of one of these variables change, so do the binary patterns.

In the last years, machine learning and big data techniques have been used to build tools aimed at improving software construction. The "big code" research line is focused on using massive code-bases from open source-code repositories to train predictive models. With this approach, different software development tools have been built, such as deobfuscators, automatic source-code translators, and error and vulnerability detectors. This dissertation poses how to use these kinds of techniques to tackle the undecidable problem of decompilation.

This dissertation proposes a method that, using supervised machine learning, improves the high-level semantic information inferred by existing decompilers. First, it identifies the variables that influence the models to be created. Different models are created, using the appropriate one depending on the values of the influencing variables. Another challenge faced by the method is the strong variability of binary code. Together with a domain expert, supervised machine learning is used to generalize the binary patterns and reduce the high dimensionality of the problem. Moreover, another iterative process is defined to assist the expert in the creation of decompilation models for different language constructs. Therefore, machine learning is not only used to train the models, but also as a feature engineering method.

We design and implement a platform for the automatic creation of

datasets to build predictive models. It facilitates the instrumentation of source code that helps to associate binary patterns and high-level language constructs. Binary patterns are extracted and generalized (as explained above) to reduce the number of features and improve the performance of the models. The datasets are then created, identifying the binary patterns for each instance (individual or row), and labeled with the target high-level construct. The platform implementation has been highly parallelized, obtaining 3.5 factors of speedup out of a theoretical maximum value of 4 factors (in a 4-core multiprocessor).

Our models are trained with C source code, but it is not easy to gather massive code-bases of standard C projects. For this reason, we implemented Cnerator, a stochastic C source-code generator. It is highly configurable, allowing the user to specify the probability distributions of each language construct, properties of the generated code, and post-processing modifications of the output programs. The synthetic code generated is added to different real open-source projects. In this way, we cover a high search space (synthetic code) and the common C idioms used by real programmers.

Using all these elements, we apply the proposed method to decompile the high-level type returned by functions. We first build the dataset with synthetic and real programs. Then, we train 14 different models using traditional machine learning algorithms. Those models are evaluated with 3 distinct methods and compared with the state-of-the-art decompilers. All the models outperform the existing decompilers, for the 3 evaluation methods. The best predictive model obtains a 79.1% F_1-score, whereas the best decompiler gets a 30% F_1-score.

The dataset created is also used to document the binary patterns used to predict the high-level return type. We create association rules that correlate binary patterns with C return types. Such patterns combine the binary code used to return a value (inside the function body), and the binary patterns that use the returned expression after the function is called. This information is a valuable asset to improve the implementation of existing decompilers.

Keywords

Big code, decompilation, machine learning, binary patterns, language constructs, assembly patterns, big data, Cnerator

Resumen

La decompilación es el proceso dentro de la ingeniería inversa de software encargado de recuperar el código fuente de alto nivel a partir de un fichero binario. Aunque el código fuente obtenido tiene el mismo comportamiento que el programa original, su legibilidad suele ser inferior y casi siempre es distinto al código escrito por el programador. La obtención del programa original a partir de un binario es considerado como un problema indecidible puesto que, cuando se compila un programa, el compilador descarta mucha información de alto nivel no necesaria en el binario generado.

Para tratar de reconstruir esta información, los decompiladores asocian patrones de instrucciones ensamblador a construcciones de alto nivel, permitiendo obtener construcciones de alto nivel a partir de patrones binarios. No obstante, la obtención de los patrones binarios es una tarea compleja y éstos dependen de variables tales como el compilador utilizado, opciones de compilación, el lenguaje de alto nivel de origen y el procesador para el que se genera el binario, entre otros. Si alguna de estas variables cambia, los patrones binarios a detectar también cambian.

En los últimos años se están utilizando técnicas de aprendizaje automático y *big data* para construir herramientas orientadas a mejorar el desarrollo de software. La línea de investigación *big code* obtiene grandes volúmenes de código fuente a partir de repositorios de código fuente para entrenar modelos probabilísticos predictores. Con este tipo de técnicas se han construido herramientas de diversa índole, tales como desofuscadores de código, traductores automáticos o detectores de errores y vulnerabilidades en código fuente. Esta tesis plantea en qué medida este tipo de técnicas pueden ser utilizadas para resolver el problema indecidible de la decompilación.

Esta tesis describe un método que, utilizando aprendizaje automático supervisado, mejora la extracción de información de alto nivel en comparación con los descompiladores existentes en el mercado. Inicialmente, el método plantea la creación de distintos modelos en función de las variables influyentes en la detección de los patrones binarios, eligiendo uno de los modelos creados una vez se hayan detectado los valores de las variables influyentes. Otro de los retos abordados por el método es la enorme variabilidad en la representación binaria de instrucciones ensamblador. Para ello, se define proceso de generalización de patrones en el que el aprendizaje automático es utilizado junto con un experto de dominio para reducir la variabilidad de dichos patrones.

Finalmente, un proceso iterativo es el encargado de ayudar al experto de en la creación de modelos que puedan inferir nuevas construcciones en lenguajes de alto nivel. El método propuesto no se limita a utilizar aprendizaje automático para entrenar los modelos, sino que también lo emplea como un mecanismo de descubrimiento de características (*feature engineering*).

Para entrenar los modelos predictivos de manera automática, diseñamos e implementamos una plataforma altamente parametrizable encargada de generar los conjuntos de datos usados en el entrenamiento. Para ello, primero permite instrumentar el código fuente de alto nivel para relacionar las construcciones de alto nivel y su representación binaria. Posteriormente se extraen estos patrones binarios iniciales y se aplican las generalizaciones previamente identificadas por el método descrito en el párrafo anterior. Finalmente se compone el conjunto de datos, caracterizando cada instancia, fila o individuo mediante los patrones binarios obtenidos y etiquetándolo con la construcción de alto nivel oportuna. La implementación ha sido altamente paralelizada y por ello es capaz de obtener una mejora de 3,5 factores sobre un máximo teórico de 4, cuando se ejecuta en un procesador con 4 núcleos.

Nuestros modelos se entrenan con código fuente C, pero es difícil obtener grandes volúmenes de código C estándar que pueda compilarse en diversos compiladores. Por ello implementamos un generador de código estocástico llamado Cnerator. Este generador, nos permite generar grandes volúmenes de código fuente con descripciones probabilísticas de las distintas construcciones del lenguaje, así como asegurar que se cumplen determinadas reglas especificadas por el usuario. El código sintético generado por Cnerator es enriquecido con código real obtenido de repositorios de código fuente. Así conseguimos cubrir un elevado espacio de búsqueda (código sintético) y representar las construcciones típicas mas utilizadas por los programadores (código real).

Con todo ello, se construyó un modelo predictivo orientado a detectar el tipo de alto nivel retornado por todas las funciones existentes en código binario. Tras construir el conjunto de datos, se crearon 14 modelos clasificadores con distintos algoritmos de aprendizaje automáticos, evaluándolos con 3 métodos que consideran su capacidad predictiva ante código no utilizado en el entrenamiento. Todos los modelos obtuvieron mejores resultados que todos los decompiladores existentes, para los 3 métodos de evaluación. Comparando la capacidad predictiva del mejor modelo y el mejor decompilador, nuestro sistema obtuvo un F_1-score del 79,1 % frente al 30 % F_1-score del mejor decompilador del mercado.

Tras la creación de los conjuntos de datos para entrenar los modelos, realizamos un análisis de los primeros para documentar los patrones utilizados en la clasificación del tipo de retorno de las funciones. Con

este fin obtuvimos y analizamos reglas que asocian patrones binarios a los tipos de retorno de alto nivel. Dichas reglas de asociación combinan patrones binarios obtenidos dentro del cuerpo de una función con patrones de utilización del valor devuelto tras las invocaciones a la función. Las reglas combinan la información binaria de cómo una función retorna un valor con información de cómo se usa dicho valor, para detectar los tipos de retorno de alto nivel. Estos patrones pueden ser incluidos en los decompiladores actuales y así mejorar la inferencia que hacen de los tipos de retorno de las funciones decompiladas.

Palabras Clave

Big code, decompilación, aprendizaje automático, patrones binarios, construcciones del lenguaje, patrones ensamblador, big data, Cnerator

Contents

Contents vii

List of Figures x

List of Tables xii

1 Introduction 1
- 1.1 Motivation 1
- 1.2 A motivating example 2
- 1.3 Contributions 4
- 1.4 Structure of the document 4

2 Related work 6
- 2.1 Type inference 6
- 2.2 Machine learning for decompilation 7
- 2.3 Other uses of machine learning for code reversing 8
- 2.4 General research on decompilation 9

3 Automatic extraction of patterns in native code 11
- 3.1 Architecture 12
  - 3.1.1 Instrumentator 13
  - 3.1.2 Binary pattern extractor 14
  - 3.1.3 Pattern generalizator 16
  - 3.1.4 Classifier 17
  - 3.1.5 Dataset generator 18
  - 3.1.6 Binary files processing 18
  - 3.1.7 Representing non-sequential patterns 19
- 3.2 Implementation 20
- 3.3 Evaluation 21
  - 3.3.1 Methodology 21
  - 3.3.2 Increasing number of modules 22
  - 3.3.3 Increasing number of workers 23
  - 3.3.4 Increasing number of cores 24
  - 3.3.5 Increasing number of modules and workers 24
  - 3.3.6 Increasing number of functions 25
  - 3.3.7 Increasing *MaxOffset* and *MaxSize* 26
  - 3.3.8 Increasing types of patterns 26

3.3.9 Execution time for a real case scenario 27

4 Method to create predictive models for decompilation 29
4.1 Challenges faced 29
4.2 Influencing variable 30
4.3 Variability of binary files 31
4.4 Language constructs 35

5 Decompilation of function return types 37
5.1 System overview 37
5.1.1 Instrumentation 39
5.1.2 Compilation 39
5.1.3 Pattern extraction 40
5.1.3.1 Extraction of binary fragments 40
5.1.3.2 Pattern generalization 41
5.1.3.3 Dataset creation 41
5.2 Methodology 41
5.2.1 Data origin 41
5.2.2 Data size 42
5.2.3 Classification 42
5.2.4 Feature selection 42
5.2.5 Hyperparameter tuning 43
5.2.6 Evaluation of model performance 43
5.2.7 Selected decompilers 44
5.2.8 Data analysis 45
5.3 Evaluation 45
5.3.1 Grouping types by size 46
5.3.1.1 Data size 47
5.3.1.2 Feature selection 48
5.3.1.3 Hyperparameter tuning 49
5.3.1.4 Results 49
5.3.2 Classifying with high-level types 51
5.3.2.1 Results 53

6 Binary patterns found 56
6.1 Association rules 56

7 Controlled stochastic generation of standard C source code 62
7.1 Software framework 62
7.1.1 Software functionalities 62
7.1.2 Architecture 63
7.2 Illustrative example 64

8 Conclusions 68

9 Future work 70
9.1 Automatic detection of influencing variables 70
9.2 Decompilation of other native languages 70

9.3 Inference of other language constructs . . . 71
9.4 Graph neural networks . . . 71
9.5 Addition of patterns to existing decompilers . . . 72
9.6 Adding additional knowledge with inductive logic programming . . . 73

A Association rules found **74**
A.1 Class `bool` . . . 75
A.2 Class `char` . . . 76
A.3 Class `short` . . . 76
A.4 Class `int` . . . 77
A.5 Class `pointer` . . . 78
A.6 Class `struct` . . . 80
A.7 Class `long long` . . . 80
A.8 Class `float` . . . 80
A.9 Class `double` . . . 81
A.10 Class `void` . . . 81

B Binary pattern generalizations **82**

C Cnerator user manual **87**
C.1 Installation . . . 87
C.2 Usage . . . 87
C.3 Command line arguments . . . 88
C.4 Syntactic constructs . . . 89
C.5 Probability specification files . . . 92
C.6 Function generation files . . . 92
C.7 Post-processing specification files . . . 93

D Hyperparameters **95**

E Publications **97**

References **98**

List of Figures

1.1 C source code example. 2

3.1 Platform architecture, receiving high-level code. 12
3.2 Instrumentation rule for `return` statements. 13
3.3 Instrumentation rule for procedures. 13
3.4 Individual detector to recognize functions. 14
3.5 Pattern detector rule to recognize RET patterns. 15
3.6 Pattern detector rule to recognize POST CALL patterns. 15
3.7 Platform architecture, receiving high-level code. 16
3.8 Pattern generalization rule of *move* instructions. 17
3.9 Classification rule associating each function with its return type. . 17
3.10 Platform architecture to process binary code. 18
3.11 Parallelization of the platform implementation. 20
3.12 Execution time for an increasing number of modules. 23
3.13 Execution time for an increasing number of workers. 24
3.14 Execution time for an increasing number of cores. 24
3.15 Execution time for an increasing number of modules and workers. 25
3.16 Execution time for an increasing number of functions. 25
3.17 Execution time per function, increasing the number of functions. . 26
3.18 Execution time for an increasing number of *MaxOffset* and *MaxSize*. 27
3.19 Execution time when extracting different types of patterns. 28

4.1 Variables that influence the decompilation predictive model. . . . 31
4.2 Feature engineering process. 32

5.1 System architecture. 38
5.2 Original C source code (left) and its instrumented version (right). 39
5.3 A RET binary fragment/chunk 40
5.4 The left-side code is transformed by `cl` into the right-side code to allow returning structs in `eax`, which are actually passed to the caller as pointers. 46
5.5 Classifiers accuracy for increasing number of functions (classifiers of types with different size and representation). 47
5.6 Coefficient of variation of the last 10 accuracy values in Figure 5.5 (classifiers of types with different size and representation). 48
5.7 Accuracies of our classifiers and the existing decompilers, using the three different evaluation methods described in Section 5.2.6 (classifiers of types with different size and representation). 49

5.8 Accuracy of a decision tree for different percentage of real functions included in the training dataset (classifiers of types with different size and representation). The red dot indicates the value where the CoV of the last 10 accuracies is lower than 1 %. 50
5.9 Classifiers accuracy for increasing number of functions (classifiers of high-level types). 52
5.10 Accuracies of our classifiers and the existing decompilers, using the three different evaluation methods described in Section 5.2.6 (classifiers of high-level types). 53
5.11 Accuracy of a decision tree for different percentage of real functions included in the training dataset (classifiers of high-level types). The red dot indicates the value where the CoV of the last 10 accuracies is lower than 1 %. 53

7.1 Architecture of Cnerator. 63
7.2 Two example JSON files used to customize program generation with Cnerator. The left-hand side shows a sample probability specification file, and the right-hand side specifies an example of controlled function generation. 65
7.3 Python code excerpt of an AST post-processing example. 66

9.1 Graph neural network for inferring the high-level type returned by functions. 72

List of Tables

4.1 Generalization examples made by our system. *reg* variables represent registers, *literal* integer literals, *address* absolute addresses, **address* absolute addresses dereferencing and *offset* relative addresses. 34

5.1 Example dataset generated by our system. 38
5.2 Open-source C projects used. 41
5.3 Relationship between the target variable (types grouped by size and representation) and the C high-level types. 46
5.4 Best feature selection method used for each classifier (classifiers of types with different size and representation). 48
5.5 Performance of the classifiers and existing decompilers using the third evaluation method (classifiers of types with different size and representation). 95 % confidence intervals are expressed as percentages. Bold font represents the best values. If one column has multiple cells in bold, it means that values are not significantly different. 50
5.6 Confusion matrix for the decision tree classification of high-level types, with a balanced dataset comprising 6,000 functions. 51
5.7 Best feature selection method used for each classifier (classifiers of high-level types). 52
5.8 Performance of the classifiers and existing decompilers using the third evaluation method (classifiers of types with different size and representation). 95 % confidence intervals are expressed as percentages. Bold font represents the best values. If one column has multiple cells in bold, it means that values are not significantly different. 54
5.9 Confusion matrix of the model in Table 5.6, including the generalizations of Appendix B. 54
5.10 Performance gains obtained for high-level type classification, when the generalizations in Appendix B are included in the dataset. . . 55

6.1 Example association rules obtained from the dataset. *reg* variables represent registers, *literal* integer literals, *address* absolute addresses, **address* absolute addresses dereferences and *offset* relative addresses. 59
6.2 Binary encodings of `fld` and `fstp` instructions. 60

Chapter 1

Introduction

1.1 Motivation

Decompilers are tools that receive binary code as an input and generate high-level code with the same semantics as the input. Although the decompiled source code can be recompiled to produce the original binary code, the high-level source code is not commonly the one originally written by the programmer. In fact, the source code is usually much less readable than the original one [1]. This is because obtaining the original source code from a binary file is an undecidable problem [1]. The cause is that the compiler discards high-level information in the translation process, such as type information, that cannot be recovered in the inverse process.

In the implementation of current decompilers, experts analyze source code snippets and the associated binaries generated by different compilers to identify decompilation patterns. Such patterns associate sequences of assembly instructions with high-level code constructs. These patterns are later included in a decompiler implementation [2, 3]. The identification of these code generation patterns is not an easy task, because of many factors such as the optimizations implemented by compilers, the high expressiveness degree of high-level languages, the compiler used, the target CPU, and the compilation parameters.

The use of large volumes of source code has been used to create tools aimed at improving software development [4]. This approach has been termed "big code" [5] since it applies big data techniques to source code. In the big code area, existing source-code corpora have already been used to create different systems such as JavaScript deobfuscators [6], automatic C#-to-Java translators [7], and tools for detecting program vulnerabilities [8]. Probabilistic models are built with machine learning and natural language processing techniques to exploit the abundance of patterns in source code [9].

Our idea is to use large portions of high-level source code and their related binaries to train machine learning models. These models will help us find code generation patterns not used by current decompilers. Machine learning has already been used for decompilation (Chapter 2). Different recurrent neural net-

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

struct stats { int count; int sum; int sum_squares; };

void stats_update(struct stats * s, int x, bool reset) {
  if (s == NULL) return;
  if (reset) * s = (struct stats) { 0, 0, 0 };
  s->count += 1;
  s->sum += x;
  s->sum_squares += x * x;
}

double mean(int data[], size_t len) {
  struct stats s;
  for (int i = 0; i < len; ++i)
    stats_update(&s, data[i], i == 0);
  return ((double)s.sum) / ((double)s.count);
}

void main() {
  int data[] = { 1, 2, 3, 4, 5, 6 };
  printf("MEAN = %lf\n", mean(data, sizeof(data) / sizeof(data[0])));
}
```

Figure 1.1: C source code example.

works have been used to recover the number and the built-in types of function parameters [10]. Extremely randomized trees and conditional random fields have provided good results inferring basic type information [11]. Decompilation has also been tackled with encoder-decoder translation neural networks [12] and with a genetic programming approach [13] (these works are detailed in Chapter 2)).

In this dissertation, we propose a method to use supervised machine learning to improve the high-level information inferred by decompilers (Chapter 4). Particularly, we apply the proposed method to improve the performance of existing decompilers in predicting the types returned by functions in high-level programs. For that purpose, we implement an infrastructure that instruments C source code to label binary patterns with high-level type information (Chapter 3). That labeled information is then used to build predictive models (Chapter 5). Moreover, the dataset created is used to document the binary patterns found by the classifiers and facilitate its inclusion in the implementation of any decompiler (Chapter 7).

1.2 A motivating example

Before detailing the objectives of this dissertation, we describe a motivating example related to the recovery of high-level type information.

Figure 1.1 shows an example C program compiled with the Microsoft's `cl` 32-bit compiler. The generated binaries were then decompiled with four different decompilers. The following function signatures were produced by those decom-

pilers:

- Signature of the `stats_update` function:
  - IDA Decompiler: `int __cdecl sub_401000(int a1, int a2, char a3)`
  - RetDec: `int32_t function_401000(int32_t * a1, int32_t a2, int32_t a3)`
  - Snowman: `void ** fun_401000(void ** ecx, void ** a2, void ** a3, void ** a4, void ** a5, void ** a6)`
  - Hopper: `int sub_401000(int arg0, int arg1, int arg2)`
- Signature of the `mean` function:
  - IDA Decompiler: `int __cdecl sub_4010A0(int a1, unsigned int a2)`
  - RetDec: `int32_t function_4010a0(int32_t a1, uint32_t a2)`
  - Snowman: `void ** fun_4010a0(void ** ecx, void ** a2, void ** a3)`
  - Hopper: `int sub_4010a0(int arg0, int arg1)`

For both functions, no decompiler infers the correct return type. Although IDA Decompiler, RetDec and Hopper detect the correct number of arguments, most of the parameter types are not correctly recovered by any decompiler[1].

As mentioned, we take as a particular case scenario the inference of the high-level type returned by functions, by analyzing their binary code. This is a complex task because the value is returned to the caller in a register (`bool`, `char`, `short`, `int`, `long`, `pointer`, and `struct`[2] values are returned in the accumulator; `long long` in `edx:eax`; and `float`, `double` and `long double` in the FPU register stack), but the value stored in that register could be the result of a temporary computation in a function returning `void`. We search for patterns in the binary code before returning and after the invocation, to see if we can recover the high-level return type written by the programmer.

The problem to be solved is a multi-label classification problem, where the target variable is an enumeration of all the high-level built-in types of the C programming language (including `void`), plus the type constructors that can be returned[3] (`pointer` and `struct`). The models we build (Chapter 4) are able to correctly infer the high-level C types returned by the `stats_update` and `mean` functions in Figure 1.1.

[1] `size_t` and `uint32_t` are aliases for `unsigned int` in a 32-bit architecture.

[2] A `struct` is commonly returned as a pointer to `struct` (*i.e.*, its memory address is returned instead of its value).

[3] In C, a function returning an array actually returns a pointer. For Microsoft's `cl`, the `union` type is actually represented as `int`, `long` or `struct`, depending on its size (explained in Section 5.3.1).

1.3 Contributions

1. Improving the high-level information inferred by decompiles using supervised machine learning. Particularly, we improve the inference of the types returned by functions. The models are trained with a dataset that relates binary patterns with high-level return types. To evaluate their performance, we conduct a comparison with four different decompilers.

2. Design and implementation of a platform for the automatic extraction of patterns from binary files (Chapter 3). The implemented platform is highly parameterized so that it can be used in different scenarios. When debug information is not available or the platform is used directly with binary code, it can be used to predict features of native code. The implementation has been parallelized (data and task parallelization) to provide important runtime performance benefits for multicore architectures.

3. A method to create predictive models for improving the performance of existing decompilers (Chapter 4). The proposed method considers all the variables that influence the decompilation process, the huge variability of binary instructions and the different language constructs provided by high-level programming languages. Machine learning is used not only to create the predictive modes but also as a feature engineering technique.

4. A collection of patterns to identify the high-level types returned by a function from its binary code (Chapter 6). The dataset created is used to obtain explainable patterns that predict return types. Such patterns combine binary sequences of code used to return an expression and the code written just after function invocation. They could be included in existing open-source decompilers.

5. A stochastic C source code generator tool for training big code machine learning models (Chapter 7). This tool, called Cnerator, creates large amounts of standard ANSI/ISO C source code [14]. Moreover, it is highly customizable to generate all the syntactic constructs of the C language, necessary to build accurate predictive models with machine learning algorithms.

1.4 Structure of the document

This PhD dissertation is structured as follows. The next chapter describes the related work. Chapter 3 describes the platform used to extract the binary patterns and generate the datasets. The method used to create predictive models from binary code is described in Chapter 4. In Chapter 5, we apply the proposed method to the problem of inferring function return types, evaluate its performance and compare it with the existing decompilers. Chapter 6 discusses some interesting patterns discovered from the dataset. Our stochastic C code generator is described in Chapter 7. Conclusions are presented in Chapter 8 and Chapter 9 identifies future lines of work.

Appendix A shows a complete set of association rules that correlate binary

patterns with high-level return types. All the generalization templates used during the dataset generation are listed in Appendix B, and the usage of Cnerator is detailed in Appendix C. In Appendix D you can see the hyperparameters selected for each predictive model. Finally, Appendix E enumerates the publications derived from this dissertation.

Chapter 2

Related work

We detail the research work related to this dissertation. We start by describing those works aimed at inferring types from binary code. Then, we see how machine learning has been used to improve decompilers, and some other uses of machine learning for code reversing. The last subsection describes the existing research on decompilation, from a general point of view.

2.1 Type inference

There are some research works aimed at inferring high-level type information from binary code. Chua *et al.* [10] use recurrent neural networks (RNN) [15] to detect the number and type of function parameters. First, they transform each instruction into word embeddings (256 double values per instruction) [16]. Then, a sequence of instructions (vectors) is used to build 4 different RNNs for counting caller arguments, counting function parameters, recovering types of caller arguments and recovering types of function parameters. In this work, they infer seven different types: `int`, `float`, `char`, `pointer`, `enum`, `union` and `struct`. They only consider `int` for integer values and `float` for real ones. They achieved 84% accuracy for parameter counting and 81% for type recovery.

He *et al.* [11] build a prediction system that takes as input a stripped binary and outputs a new binary with debug information that includes type information. They combine extremely randomized trees (ERT) with conditional random fields (CRFs) [17]. The ERT model aims to extract identifiers. Although identifiers are always mapped to registers and memory offsets, not every register and memory offset stores identifiers. Then, the CRF model predicts, the name and type of the identifiers discovered by ERT. They use a maximum a posteriori (MAP) estimation algorithm to find a globally optimal assignment. This tool handles 17 different types, but it lacks floating-point types support. This is because the library used to handle the assembly code, Binary Analysis Platform (BAP) [18], does not support floating-point instructions. Their system achieves 68.8% precision and 68.3% recall.

Mycroft [19] proposes a type reconstruction algorithm that uses unification to recover types from binary code. Mycroft starts by transforming the binary code

into a register transfer language (RTL) intermediate representation. The RTL representation is then transformed into a single static assignment (SSA) form to undo some optimizations performed by the compiler. Then, each code instruction is used to generate constraints about the type of its operands, regarding their use. Those constraints are used to recover types by applying a modified version of Milner's W algorithm [20]. In this variation, any constraint violation causes type reconstruction (recursive structs and unions) instead of premature termination. This work does not discuss stack-based variables, only register-based ones.

TIE (Type Inference on Executables) [21] addresses the stack-based variable limitation of Mycroft's approach by using a variation of the value set analysis (VSA) [22] called DVSA. Additionally, the constraint solving algorithm uses subtyping to model the inherent polymorphism in assembly instructions. For example, add can be used to add two numbers but also a number and a pointer.

All these works are valid examples of static analysis, however, there exist alternatives based on gathering information dynamically, when the native code is being run [23]. Raman *et al.* [24] use an execution trace to identify recursive data structures linked by pointers. They track all the memory allocations to identify the heap objects and use the links between them to create a shape graph. This dynamic analysis approach can be combined with static analysis to obtain finer-grain information. Caballero *et al.* [25] combine static analysis with several execution traces, being able to extract the prototype of target functions.

2.2 Machine learning for decompilation

Some works are not focused on type inference exclusively, but they undertake decompilation as a whole, including type inference. The works in [12, 26, 27] propose different systems based on neural machine translation [28, 29]. They use RNNs with an encoder-decoder scheme to learn high-level code fragments from binary code, for a given compiler.

Katz (Deborah) *et al.* [12] use RNN models for snippet decompilation with additional post processing techniques. They tokenize the binary input with a byte-by-byte approach and the output with a C lexer. C tokens are ranked by its frequency and replaced by the ranking position. Less frequent tokens (below a frequency threshold) are replaced by a common number to minimize the vocabulary size. This transformation reduces the number of tokens, speeding up the training of the RNN. For the binary input, a language model is created, and byte embeddings are found for the binary information. Once the encoder-decoder scheme is trained, translation from binary code into C tokens is performed. The final step is to apply several post processing transformations, such as deleting extra semicolons, adding missing commas, and balancing brackets, parenthesis, and curly braces.

The previous work was later modified by Katz (Omer) *et al.* to reduce the compiler errors in the output C code [26]. In this previous work, most of the output C code could not be compiled because errors were found. Therefore, they modified the decoder so that it produces prefixed templates to be filled. This idea is in-

spired by delexicalization [30]. Delexicalized features are n-gram features where references to a particular slot or value are replaced with a generic symbol. In this way, locations in the output C source code are substituted with placeholders. After the translation takes place, those placeholders are replaced with values and constants taken from the binary input, improving the code recovery process up to 88%.

Coda [27] is an end-to-end neural-based framework for code decompilation. First, Coda employs an instruction type-aware encoder and a tree decoder for generating an abstract syntax tree (AST) with attention feeding during a code sketch generation stage. Afterward, it updates the code sketch using an iterative error correction machine guided by an ensembled neural error predictor. An approximate candidate is found first and then fixed to produce a compilable program that generates the original binaries when compiled. Evaluation results show that Coda achieves 82% program recovery accuracy on unseen binary samples.

Schulte *et al.* [13] propose genetic programming to generate readable C source code from compiled binaries. Taking binary code as the input, an evolutionary search seeks a combination of source code excerpts from a big code database. That source code is compiled into an executable, which should be byte-equivalent to the original binary. The decompiled source code reproduces the behavior, both intended and unintended, of the original binary. As they use evolutionary search, decompilation time can vary dramatically between executions.

2.3 Other uses of machine learning for code reversing

Apart from recovering high-level type information from binaries, other works use machine learning for different code reversing purposes [31]. Rosenblum *et al.* [32] use CRFs to detect function entry points (FEPs). They use n-grams of the generalized instructions surrounding FEPs, together with a call graph representing the interaction between FEPs. The FEP detection problem consists in finding the boundaries of each function in the binary code. CRFs allow using both sources of information together. Since standard inference methods for CRFs are expensive, they speed up training with approximate inference and feature selection. Nonetheless, feature selection took 150 days of computation on 1171 binaries. This approach does not seem to be tractable for big code scenarios.

Bao *et al.* [33] utilize weighted prefix trees (or weighted tries) [34] to detect FEPs, considering generalized instructions as tree nodes. Once trained, each node represents the likelihood that the sequence from the root node to the current node will be a FEP. They trained the model with 2064 binaries in 587 computing hours, obtaining better results than [32]. The approach of Shin *et al.* [35] uses RNNs for the same problem. The internal feedback loops of RNNs make them suitable to handle sequences of bytes. This approach reduces training time to 80 computing hours using the same dataset as Bao *et al.* [33], while performing slightly better.

Rosenblum *et al.* [36] use CRFs to detect the compiler used to generate bi-

nary files. Binaries frequently exhibit gaps between functions. These gaps may contain data such as jump tables, string constants, regular padding instructions and arbitrary bytes. While the content of those gaps sometimes depends on the functionality of the program, they sometimes denote compiler characteristics. A CRF model is built to exploit this difference among binaries and detect the compiler used to generate the binaries. They later extend this idea to detect different compiler versions and optimization levels [37].

Malware detection is another field related to binary analysis where machine learning has been used [38]. Alazab *et al.* [39] separate malware from benign software by analyzing the sequence of Windows API calls performed by the program. First, they process the binaries to extract all the Windows API invocations. Then, those sequences are vectorized and used to build eight different supervised machine learning models. Those models are finally evaluated, finding that support vector machine (SVM) with a normalized poly-kernel classifier is the method with the best results. SVM achieves 98.5% accuracy.

Rathore *et al.* [40] detect malware by analyzing opcodes frequency. They use various machine learning algorithms and deep learning models. In their experiments, random forest outperforms deep neural networks. Static analysis of the assembly code is used to generate multi-dimensional datasets representing opcode frequencies. Different feature selection strategies are applied to reduce dimensionality. They collect binaries from different sources, selecting 11,688 files with malware and 2,819 benign executables. The dataset is balanced with adaptive synthetic (ADASYN) sampling. Random forest obtained 99.78% accuracy.

2.4 General research on decompilation

There has been a long line of research on decompilation. Cifuentes [2] establishes the foundations of modern decompilers. She covers a wide range of techniques, like data-flow analysis and control-flow analysis. All this knowledge is included in the implementation of DCC [41], a decompiler for Intel 80286 DOS programs. DISC [42] is another decompiler that works with DOS executables, although it is exclusively focused on binaries generated with the Turbo C compiler.

Following the steps of Cifuentes, van Emmerick explores the use of single static assignment (SSA) form to facilitate decompilation tasks such as dead code propagation and type analysis [3]. Those techniques are included in the Boomerang decompiler [43]. The design of Boomerang is similar to DCC, but with a particular emphasis on modularity. EiNSTeiN is another open-source decompiler based on the work of van Emmerick [44].

The REC decompiler is derived from Cifuentes' work [45]. It adds the implementation of complex algorithms to analyze control flow graphs, which provide the reconstruction of some advanced constructs such as `switch` statements. Its last version also includes SSA transformations.

Hex-Rays IDA Decompiler [46] is the de facto industry standard. It derives from Guilfanov's early work [47, 48] on the Belgian company DataRescue. It is

developed as a plugin for the Interactive Disassembler Pro (IDA Pro). Since it is a proprietary development, little is known about its internals. One of its key features is the Fast Library Identification and Recognition Technology (FLIRT) [49]. FLIRT is a binary pattern-recognition library that is used to identify the compiler used to generate the binaries and to detect start-up and statically-linked code.

Troshina *et al.* implement TyDec [50] and its successor, SmartDec [51]. Both decompilers are focused on recovering specific constructs of C++ such as objects, class hierarchies and the indirect calls resulting from virtual inheritance. Snowman is a fork of SmartDec that is actively maintained [52].

Hopper [53] is a commercial decompiler developed by Cryptic Apps. It is originally aimed at decompiling Objective-C constructs such as selectors and message passing. It is available for and Mac OS X. However, it also supports decompilation of C code (as Objective-C is a superset of C) from Windows executables.

Schwartz *et al.* [54] present Phoenix, which is built on top of BAP and TIE tools from the same research group. BAP (Binary Analysis Platform) [18] transforms native assembler code to an Intermediate Language (IL) representation, TIE (Type Inference on Executables) [21] is aimed at recovering high-level type information from IL. Then, the decompiled code is tested with the original unit tests to check that decompilation preserves semantics.

RetDec (Retargetable Decompiler) [55] was first implemented by Křoustek as part of his PhD [56]. RetDec is independent of any architecture, language or ABI (Application Binary Interface) because it uses the LLVM [57] Intermediate Representation (IR) across all the phases of the decompilation. Since December 2017, RetDec is supported by AVAST.

Yakdan *et al.* developed the DREAM [58] and DREAM++ [59] decompilers, both aimed at producing goto-free code. They define a set of semantics-preserving code transformations, capable of transforming unstructured control flow graphs into structured ones.

Chapter 3

Automatic extraction of patterns in native code

The objective of obtaining high-level information from massive binary codebases demands an efficient mechanism to extract binary patterns [60]. Example tools that extract patterns in binary code are decompilers, packed executable file recognizers, authorship analyzers, and malware detectors. In this chapter, we identify the requirements of such a platform to facilitate the extraction of binary patterns.

A platform to extract binary patterns should be able to use the debug information (if any) generated by the compiler. This information is very valuable to extract such binary patterns, which could later be used by a machine-learning algorithm to create predictive tools. A large number of patterns may be extracted from a small binary program since the number of assembly instructions is much higher than in the original source high-level program. Therefore, the need of processing debug information, plus the potentially huge number of patterns to be extracted, makes it critical to use a highly parallelized and efficient approach for extracting the patterns.

The platform should also be highly parameterized. The individuals or instances (*i.e.*, rows in the dataset generated) to be extracted will be specified by the user. For instance, we may be interested in finding patterns for functions, snippets, function entry points, or specific regions of binary code. The same parameterization is also required to specify the features of each individual (columns in the dataset). For example, we may define the feature *mov* reg_{ax}*, any* to represent the occurrence of an assembly instruction that moves any value to the accumulator register (`ah`, `al`, `ax`, `eax` or `rax`). If that pattern (feature or column) occurs in one given region of binary code (individual, instance or row), then the corresponding value in the dataset (row and column) will be 1 [61].

The traditional method to extract features from binary code is to identify a syntactically fixed unit of code, such as functions or basic blocks, and extract the binary code inside them [33]. However, pattern extraction does not always follow this scheme. Sometimes, nonsequential patterns such as subgraphs of control flow and data dependency graphs need to be extracted. In these cases, a binary pattern extraction platform should allow the association of patterns to pieces of

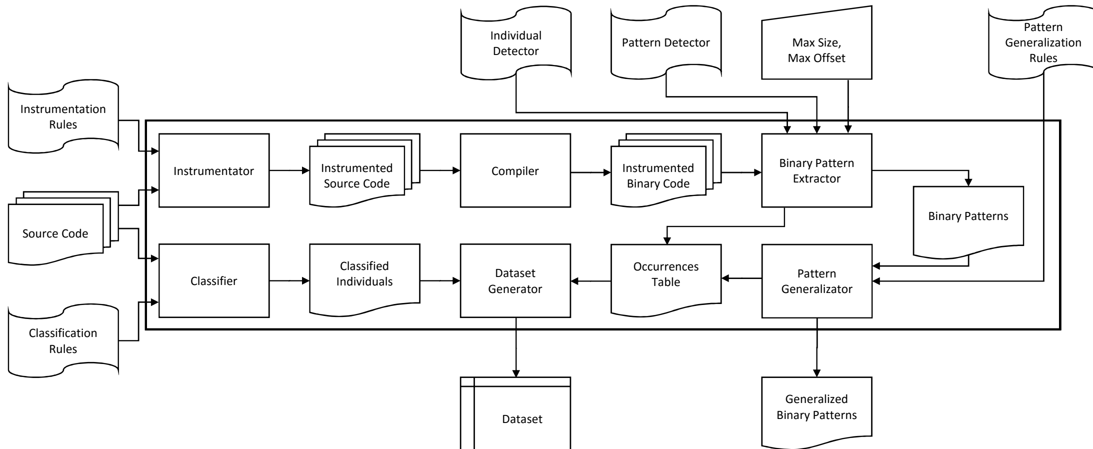


Figure 3.1: Platform architecture, receiving high-level code.

code outside their basic blocks, representing subgraph structures (Section 3.1.7). Another scenario where the traditional method is not sufficient is the analysis of binary code between two memory addresses, where the inconsistency overlapping problem caused by the variable-length instruction set must be tackled [32]. This kind of binary code analysis has been used for different purposes such as function entry points detection [32], compiler recognition [36], authorship attribution [37], and malware detection [62, 63]. Therefore, a generic platform for pattern extraction must be flexible enough to support any binary pattern extraction method (not just the traditional one) and reduce development and execution times.

We first present the architecture of the implemented platform for the automatic extraction of patterns in native code (Section 3.1). The platform is highly parameterized so that it could be used in different scenarios. Its parallel implementation provides important runtime performance benefits when multicore architectures are used. It also uses the debug information that may be provided by a compiler. The extracted patterns may be used by other tools for different purposes. After the architecture, we present the most important challenges faced in the implementation (Section 3.2). Section 3.3 evaluates the platform, measuring the execution time of different configurations for a large number of programs. We will see how the parallelization provides significant performance improvements, and its efficiency is maintained for big volumes of programs. In Chapter 5, we use it for the particular problem of predicting the high-level types returned by functions.

3.1 Architecture

Our platform has two working modes. The most versatile is the one shown in Figure 3.1. The platform receives the high-level source program that will be used to generate the binary application. In this mode, the platform allows instrumenting the input program and it uses the debug information produced by the compiler. When the high-level program is not available, we provide another configuration to process binary files, described in Section 3.1.6.

```
procedure FUNCTION_RETURN_INSTRUMENTATION(program)
    for all stmt in program do
        if stmt is type_return id_func ( (type_arg id_arg) * ) stmt_body * then
            labels ← 1
            for all stmt in stmt_body * do
                if stmt is return exp then
                    stmt ← __RETURN_id_func_labels__ : stmt
                    labels ← label + 1
```

Figure 3.2: Instrumentation rule for `return` statements.

```
procedure PROCEDURE_RETURN_INSTRUMENTATION(program)
    labels ← 1
    for all stmt in program do
        if stmt is type_return id_func ( (type_arg id_arg) * ) stmt_body *
          ↪ and type_return = void then
            stmt_body * ← stmt_body * ; __RETURN_id_func_labels__ : return;
            labels ← labels + 1
```

Figure 3.3: Instrumentation rule for procedures.

3.1.1 Instrumentator

This module allows code instrumentation of the high-level input program. The objective is to add information to the input program so that it will be easier to find the patterns in the corresponding binary code generated by the compiler. It can also be used to delimitate those sections of the generated binary code we want to extract patterns from (Section 3.1.2), ignoring the rest of the program. Notice that, once the machine learning model has been trained with the dataset generated by the platform, the binary files passed to the model will not include that instrumented code. Therefore, the instrumentation module should not be used to extract patterns that cannot be later recognized from stripped binaries.

This module traverses the Abstract Syntax Tree (AST) of the *Source Program* and evaluates the *Instrumentation Rules* provided by the user. While traversing the AST, if the precondition of one instrumentation rule is fulfilled, its corresponding action is executed. The action will modify the AST with the instrumented code, producing the input to be passed to the compiler (next module in Figure 3.1).

Figure 3.2 shows the instrumentation rule we used for the example problem of inferring the high-level types of functions (in Section 3.2 we describe how instrumentation rules are implemented). For all the `return` statements in a program, the rule adds a dummy label before the `return` statement. This label has the function identifier id_{func} followed by a consecutive number, since a function body may have different `return` statements. This label will be searched later in the binary code (using the debug information) to locate the binary instructions generated by the compiler for the high-level `return` statements. These binary instructions will be used to extract the binary patterns (Section 3.1.2).

Figure 3.3 shows another example of one instrumentation rule. Recall that

```
function INDIVIDUAL_DETECTOR(program)
    individuals ← []
    instruction ← program[0]
    repeat
        if is_function(instruction) then
            individuals ← individuals + label(instruction)
        instruction ← next(instruction)
    until not next(instruction)
    return individuals
```

Figure 3.4: Individual detector to recognize functions.

the previous instrumentation rule was aimed at locating binary instructions between a `__RETURN` label and a `ret` assembly instruction. However, C functions returning `void` usually do not have an ending `return` statement. Therefore, the instrumentation rule in Figure 3.3 adds both, the expected label and the return statement.

Adding labels is an easy way to instrument code. However, more sophisticated approaches can be used. For example, expressions may be translated into dummy function invocations that are actually used as marks to be identified in the pattern extraction phase (Section 3.1.2). Another typical approach is adding innocuous sequences of assembly instructions (*e.g.*, `nop`) to be found in the pattern extraction phase. The user must be careful when selecting the instrumentation approach, checking that the instrumented code does not produce unexpected changes to the generated binaries or the target patterns.

As shown in Figure 3.1, the *Instrumentation Rules* are used to translate *Source Code* into *Instrumented Code*. The *Instrumented Code* is then compiled, producing the *Instrumented Binary Code*.

3.1.2 Binary pattern extractor

This module performs three tasks. First, it identifies the binary code fragments or chunks representing the individuals (rows) in the generated dataset. Second, it extracts the binary patterns (columns) detected for each individual. Binary patterns are used as features to later classify the individuals. The third task is to store the individuals and patterns in an *Occurrence Table*, which will be later used to generate the final dataset. We now detail these three tasks.

The *Individual Detector* initially recognizes each individual in the binary code. It must implement a function to collect all the individuals. Figure 3.4 shows the *Individual Detector* of our example, where functions are recognized as individuals. In the figure, `is_function` returns whether the parameter is the first instruction in a function by using the debug information generated by the compiler. Once one function is detected, its label (the return type) is added to the *individuals* list.

After identifying the individuals, we must extract the binary patterns we are looking for. To this end, the user should provide a *Pattern Detector* comprising

```
function RET_PATTERN_DETECTOR(instruction)
    if instruction is not __RETURN_id_func_n__ then
        return null
    begin_instruction ← instruction
    while not instruction is ret do
        instruction ← next(instruction)
    return (id_func, ( begin_instruction, next(instruction)))
```

Figure 3.5: Pattern detector rule to recognize RET patterns.

```
function POST_CALL_PATTERN_DETECTOR(instruction)
    if instruction is call id_func then
        begin_instruction ← instruction
        for i = 0 to MaxSize + MaxOffset do
            instruction ← next(instruction)
        return (id_func, (begin_instruction, instruction))
    return null
```

Figure 3.6: Pattern detector rule to recognize POST CALL patterns.

a collection of predicate functions. These functions receive one instruction of the instrumented binary program. In case that instruction is not included in the expected pattern, `null` must be returned. If the pattern is identified, a pair containing the individual and the range of instructions in the pattern (another pair) is returned.

Figure 3.5 presents a *Pattern Detector* for our example. It recognizes the return pattern added by the *Instrumentator*. If the instruction label is `__RETURN`, the *Pattern Detector* recognizes the pattern. The corresponding function is returned as the first element of the pair. The second one is the range of instructions comprising the pattern: the first one (the one labeled `__RETURN`) and the next instruction after the following `ret`.

Figure 3.6 shows another pattern detector used in our example. It detects as a pattern the instructions after one `call` (we call it POST CALL). In this case, the individual associated with the pattern is not the function the instruction belongs to, but the function called. Similarly, we have also specified a pattern with the instructions before `call` (called PRE CALL), not shown in the figure. The idea of these two patterns is that the usage of the value returned by a function (POST CALL) and the code to push its parameters (PRE CALL) may be valuable to infer the types in the function signature (return and parameter types).

At this point, the module has three types of extracted patterns: RET patterns, including the assembly code of `return` statements; and PRE and POST CALL patterns, representing the code before and after invoking a function. Each of these patterns may include a significant number of contiguous binary instructions. However, we could be interested in a small portion of contiguous instructions inside the bigger patterns. For this reason, the *Binary Extractor Pattern* has been designed to divide the patterns found into a collection of subpatterns (different partitions of the original pattern).

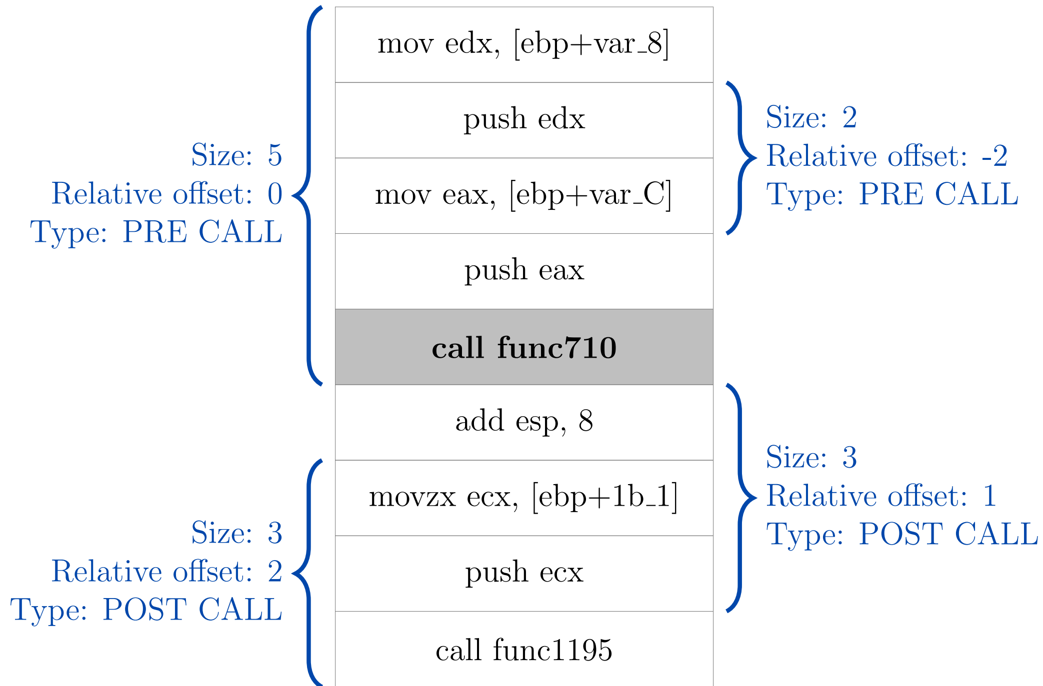


Figure 3.7: Platform architecture, receiving high-level code.

The algorithm to obtain the subpatterns is parameterized by the *MaxSize* and *MaxOffset* parameters shown in Figure 3.1. This algorithm starts with one-instruction length subpatterns (*size* = 1), increasing this value up to *MaxSize* contiguous instructions. Additionally, other subpatterns are extracted leaving offset instructions between the instruction detected by the Pattern Detector and the subpatterns. The algorithm described above (the one that increases size), was for *offset* = 0. The same algorithm is applied for *offset* = 1 and *offset* = -1 (*i.e.*, the first instruction before and after the detected instruction, which is not included in the subpattern). The absolute value of offset is increased up to *MaxOffset*.

Figure 3.7 shows four example subpatterns: PRE CALL *size* = 5 and *offset* = 0; PRE CALL *size* = 2 and *offset* = -2; POST CALL *size* = 3 and *offset* = 2; and POST CALL *size* = 3 and *offset* = 1.

The last task to be undertaken by the *Pattern Detector* is the association of the individuals with their patterns in the *Occurrences Table*. This process is done by generating as many table rows as individuals found by the *Individual Detector* (in Figure 3.4), and associating them with the rows representing each of the subpatterns found for that individual by the *Pattern Detector* functions (Figure 3.5 and Figure 3.6).

3.1.3 Pattern generalizator

Sometimes, the subpatterns found are too specific. For example, the `movzx eax, 5` and `movzx eax, 10` subpatterns are recognized as two different ones. However, they represent the information for the return type problem, which is that a literal has been moved to the accumulator register (*i.e.*, a *mov* `eax`,*literal* pattern). To

```
function MOV_GENERALIZATION(instruction)
    if instruction.mnemonic in [mov, movzx, movsx]
      ↪ and instruction.operands[1].type = register
      ↪ and instruction.operands[1].value in [eax, ax, ah, al] then
        instruction.mnemonic ← mov
        instruction.operands[2].value ← arg_any
    return (instruction, next(instruction))
```

Figure 3.8: Pattern generalization rule of *move* instructions.

```
function FUNCTION_CLASSIFICATION(program)
    individuals ← {}
    for all stmt in program do
        if stmt is type_return id_func ( (type_arg id_arg) * ) then
            individuals ← individuals[id_func ↦ type_return]
    return individuals
```

Figure 3.9: Classification rule associating each function with its return type.

this end, the objective of the *Pattern Generalizator* module is to allow the user to reduce the number of subpatterns, generalizing them (Section 4.3 discusses the generalization problem in a more detailed way).

In our platform, generalizations are expressed as Pattern Generalization Rules. As shown in Figure 3.8, those rules are implemented as functions receiving one instruction and returning their generalized pattern (or the current instruction if no generalization is required) and the following instruction to be analyzed. The example rule in Figure 3.8 generalizes the *mov* instructions that save into the accumulator register any value. The second value returned indicates the next instruction to be generalized. Its purpose is to allow the implementation of variable-instruction-length generalizations, very useful to group patterns such as the code used to pop the arguments after a function invocation.

The generalized patterns and their associations with the individuals are added to the existing *Occurrence Table* generated by the *Binary Pattern Extractor*.

3.1.4 Classifier

This module is aimed at computing the value of the classifier variable (*i.e.*, the target or the dependent variable) for each individual. The input is a representation of the high-level program; the output is a mapping between each individual and the corresponding value of the classifier variable. These associations are described by the user with the *Classification Rules*.

Figure 3.9 shows one *Classification Rule* for our example. We iterate through the statements in the program. For each function, we associate its identifier with the returned type, which is the classifier variable in our problem (we predict the return type of functions).

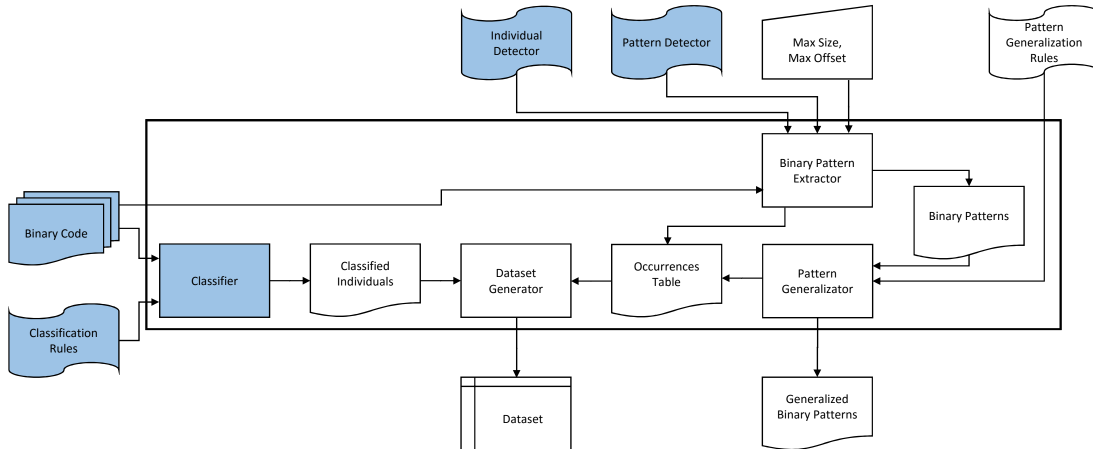


Figure 3.10: Platform architecture to process binary code.

3.1.5 Dataset generator

Finally, the *Dataset Generator* builds the dataset from the *Occurrence Table* (Section 3.1.3) and the individual classification (Section 3.1.4). One row per individual, one column per subpattern (generalized or not), and another row for the classifier variable. Cells in the dataset are Boolean values indicating the occurrence of the subpattern for that individual. Classifier (target) cells may have different categorical values. Table 5.1 shows an example dataset.

3.1.6 Binary files processing

As mentioned, the platform has two working modes. Many times, we do not have the high-level source program used to generate the native code, and we are interested in finding patterns in binary files. Different examples of this scenario include authorship, compiler and malware detection.

In order to show this second working mode of our platform, we use the research work done by Rosenblum *et al.* [32] as an example. They extract patterns from stripped binary files to detect function entry points (FEP), which existing dissemblers do not detect correctly yet [33]. Rosenblum *et al.* analyze consecutive bytes in binary files, representing them as 3-grams of assembly instructions. Once the 3-grams are extracted, they formulate the FEP identification problem as structured classification using Conditional Random Fields (CRF) [17]. An initial flat model is later enriched with the evidence that a `call` instruction indicates the existence of a FEP in the callee address. The model obtained detects FEPs more accurately than `gcc`, `icc` and `cl` compilers [32].

Figure 3.10 shows the changes to the platform architecture when we use it to process binary files, and the high-level program is not available. White elements are the same as in the previous architecture. Blue elements represent modifications of the previous working mode. All the modules related to processing high-level programs are not present.

Although the behavior of the *Binary Pattern Extractor* is the same, the rules for detecting individuals and patterns are not. The main difference is that no

instrumented code is added since the source code is not available. Depending on the case, debug information is not available either (*i.e.*, stripped binaries are used). Regarding the *Classifier* module, the *Classification Rules* must consider a plain binary file instead of a high-level program representation.

In the example of FEP detection in binary files, this is how the platform has been used to generate a dataset to create the CRF model. In the output dataset, individuals (rows) are instruction offsets in the binary file; one feature (column) will be created for each 1- 2- and 3-grams in the binary code, indicating the occurrence of that pattern in each individual; another `call` *offset* feature is added, associating that function invocation with the *offset* individual. Finally, the classifier variable (target) is 1 if the individual is a FEP and 0 otherwise (debug information is available).

In order to create the dataset described above, the *Individual Detector* creates as many individuals as instruction offsets in the binary file. The *Pattern Detector* extracts 1-, 2- and 3-grams for each offset, and a `call` feature for each different function. A `call` feature is not associated with the *offset* where the pattern is detected, but with the *offset* (memory address) being called (as done in Figure 3.6). *Pattern Generalization* is done as the normalization process described in [32]. Finally, the *Classification Rules* use the debug information to set 1 to one individual identified as a FEP and 0 otherwise.

3.1.7 Representing non-sequential patterns

In the analysis of binary applications, it is common to require the detection of non-sequential patterns, such as subgraphs of control flow and data dependency graphs. The detection of these subgraphs can be used for many different purposes, such as the FEP detection problem described in the previous subsection.

Although the *Binary Pattern Detector* module of our platform (Figures 3.1 and 3.10) is aimed at extracting patterns made up of contiguous binary instructions, the rest of the modules can be used to represent non-sequential structures such as graphs. This functionality is provided by the versatile way our platform considers the sequential patterns (features), permitting the definition of different criteria to associate these features to the corresponding individuals.

One example of this functionality is present in the decompiler scenario presented in Section 1.2. Figure 3.5 shows how RET patterns are associated with the function (individual) where the pattern was detected. In Figure 3.5, this association is represented by the first element in the tuple returned, which is the function id the `ret` instruction belongs to. Thus, the output dataset will have 1 in the cell corresponding to that function (row or individual) and pattern (column or feature). However, POST CALL patterns are associated with individuals in a different way. Figure 3.6 shows how this type of feature is not associated with the function where the pattern is detected, but with the function being called. Therefore, a machine learning algorithm trained with the generated dataset may associate non-sequential patterns (*e.g.*, there must exist a RET pattern inside the function and, in any part of the program, a POST CALL pattern invoking the

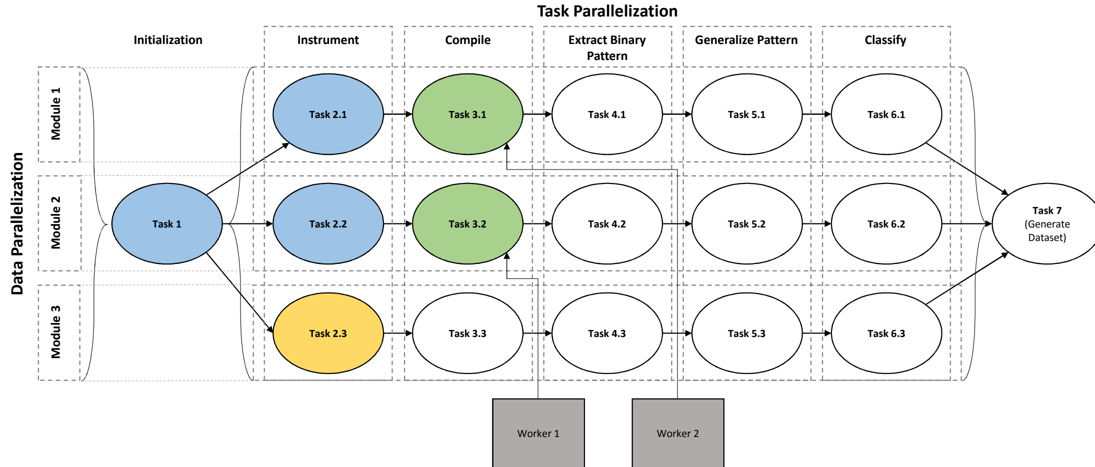


Figure 3.11: Parallelization of the platform implementation.

same function) to identify the type returned by a function.

Another example of this functionality is the FEP identification problem described in Section 3.1.6. The dataset generated by our platform can be used to create the proposed CRF model, which uses graphs for structural prediction and classification [17]. Those graphs are obtained from the dataset by using the versatile association of features to individuals already discussed.

3.2 Implementation

The *Instrumentator* and *Classifier* modules have been implemented in C++ since they use Clang [64] to process the high-level representation of C programs. The rest of the platform has been implemented in Python. We have used the disassembly services provided by IDAPython [65].

The implementation is highly parallelized, providing important performance benefits when multi-core architectures are used. The parallelization follows a pipeline scheme, where both data and task parallelism are used. Figure 3.11 shows the concrete approach followed. These have been the challenges tackled to parallelize the platform implementation:

1. Data parallelization. We identify each module in a program (`obj` files in the compiler used) as a different portion of data to work in parallel. These `obj` files can be combined in `lib` or `exe` files to produce bigger modules. In the example in Figure 3.11, three different modules are processed in parallel.

2. Task identification. The tasks to be parallelized are those identified as modules in the platform architecture (Section 3.1). As shown in Figure 3.11, an additional initialization task was defined to initialize the database and create a temporary folder where the input files are copied.

3. Task dependency. After identifying the tasks, we define the dependencies among them with a Directed Acyclic Graph (DAG). These dependencies define when two tasks can run in parallel, and when a task has to wait for

others to end. As shown in Figure 3.11, the instrumentation, compilation, binary pattern extraction, generalization and classification tasks can run in parallel. For the same piece of data, one has to wait for the previous one to finish. The initialization (at the beginning) and dataset generation (at the end) tasks cannot be parallelized. The last one waits for all the classification tasks to process all the data.

4. Task implementation. Tasks should be mapped to threads or processes. The current implementation uses the Python programming language to combine all the different modules of the architecture (implemented in Python itself or C++). Since most implementations of Python use the Global Interpreter Lock (GIL) to synchronize the execution of threads [66], we implemented tasks as processes to obtain a better runtime performance improvement with multi-core architectures [67].

5. Concurrent workers. To adapt the level of parallelization of the platform, we parameterize its implementation to run with different numbers of worker processes (Section 3.3.1). A scheduler analyzes the task DAG and tells each worker the following task to be executed. In Figure 3.11, two workers are running in parallel. Tasks 1, 2.1 and 2.2 have already been executed; Task 3.1 and 3.2 are run by, respectively, Worker 1 and 2; and Task 2.3 is the following one to be executed, once one worker is free.

6. Communication between tasks. The dependency between tasks shown in Figure 3.11 indicates that the output of one task is taken as the input of the following one. Since we implement tasks as processes, communication between them is expensive. Therefore, we implement data communication through a database, appropriately configured to obtain the expected runtime performance.

7. Task synchronization. Workers should indicate when they terminate executing one task, and the scheduler should tell them which task should be executed next. To synchronize this process, we used a `Queue` object from the `multiprocessing` module [68].

8. Tool parameterization. We configured the IDA disassembler to allow the concurrent processing of the same input file. The compilation task is represented with a Python class that can be parameterized to use different compilers, package managers, compiler options, and automating software. The external tools used to write information in the standard output (*e.g.*, the C compiler). We capture those messages and send them to a concurrent logger, adding additional information of the processes.

3.3 Evaluation

3.3.1 Methodology

The runtime performance of our platform depends on the following variables:

– The number of independent modules of compilation (or programs). Differ-

ent programs, or different modules of the same program, can be processed in parallel to create a dataset.

- The number of workers. As mentioned, the platform may run different tasks at the same time. A task is run by a worker. Depending on the number of real processors, the number of workers may produce an important benefit on runtime performance.
- The number of cores. We have run our platform with different cores of multi-core computers.
- The size of each program (or module), according to the number of individuals it may contain.
- The subpattern extraction. As described in Section 3.1.2, different subpatterns are automatically extracted from the patterns found. The *MaxSize* and *MaxOffset* parameters influence execution time.
- The number of patterns. Our platform recognizes patterns by means of the *Pattern Detector* functions specified by the user. We analyze runtime performance depending on the number of patterns defined.

We evaluate the influence of these variables on the runtime performance of the platform, and how they are related to the level of parallelization. In order to evaluate that, we fix all the variables except one and measure the runtime performance for different values of the non-fixed variable [69]. This process is repeated for all the variables.

We evaluate the platform with the example of predicting the return type in binary programs, using their C source code (the first working scheme of our system, shown in Figure 3.1). We extract RET, PRE and POST CALL patterns, divide them into different subpatterns, and perform a generalization of the subpatterns found.

The programs used for the experiments are synthetically generated by a C program generator called Cnerator [70] (Chapter 7). Cnerator provides us with a rich battery of programs. It also allows the generation of particular function signatures per module. Moreover, programs can be generated with different numbers of compilation units (modules), so that we can measure different modularizations of the very same program.

In order to be able to change the number of cores, all the tests were carried out on a Hyper-V virtual machine with 4 processors and 8GB of RAM, running an updated 64-bit version of Windows 8.1. The host computer was a 3.60 GHz Intel Core i7-4790 system with 16 GB of RAM, running an updated 64-bit version of Windows 10. The tests were executed after system reboot, removing the extraneous load, and waiting for the operating system to be loaded [71].

3.3.2 Increasing number of modules

In this first experiment, we increase the modules in a program from 1 to 8 and fix the number of cores and workers to 4. For this experiment and the following

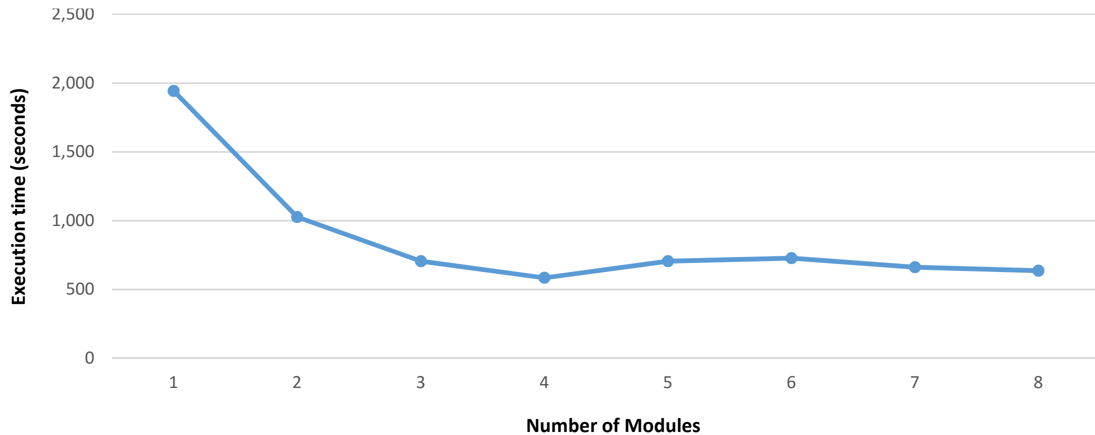


Figure 3.12: Execution time for an increasing number of modules.

ones, the value of *MaxSize* is 4 and *MaxOffset* is 0. We extract RET, PRE and POST CALL patterns.

The program to be analyzed has 10,000 functions (individuals), so we have 1 module with 10,000 functions, 2 modules with 5,000 functions, and so on, up to 8 modules with 1,250 functions. Therefore, all the configurations have the same dataset with 10,000 functions, and the same program is processed.

Figure 3.12 shows the benefits of parallelization for an increasing number of modules. The execution time of processing the same program drops when the number of modules is increased until 4 modules (the number of cores and workers). At that point, the platform processes the program 3.33 times faster than the same program with one module (*i.e.*, the sequential implementation). According to Amdahl's law, the maximum theoretical performance benefit that could be obtained for that configuration is 4 factors [72].

For more than 4 modules, there is no significant benefit because this configuration has 4 cores. Figure 3.12 shows that is not a significant penalty for 8 programs when the number of programs is higher than the number of cores. The slight worsening for 5, 6 and 7 programs is caused by the selection of 4 workers and cores. After processing 4 programs in parallel, the processing of the fifth one makes the rest of the workers wait for completion, causing a slight performance drop.

3.3.3 Increasing number of workers

In this case, the number of workers goes from 1 to 8, fixing the number of cores and modules to 4. Each module has 2,500 functions (10,000 for the whole program).

Figure 3.13 shows how execution time is reduced as the number of workers increases. With 4 workers, the platform reaches the lowest value, 3.5 times faster than the sequential execution. For 5 workers or more, there is no benefit because those extra workers keep waiting for tasks to end.

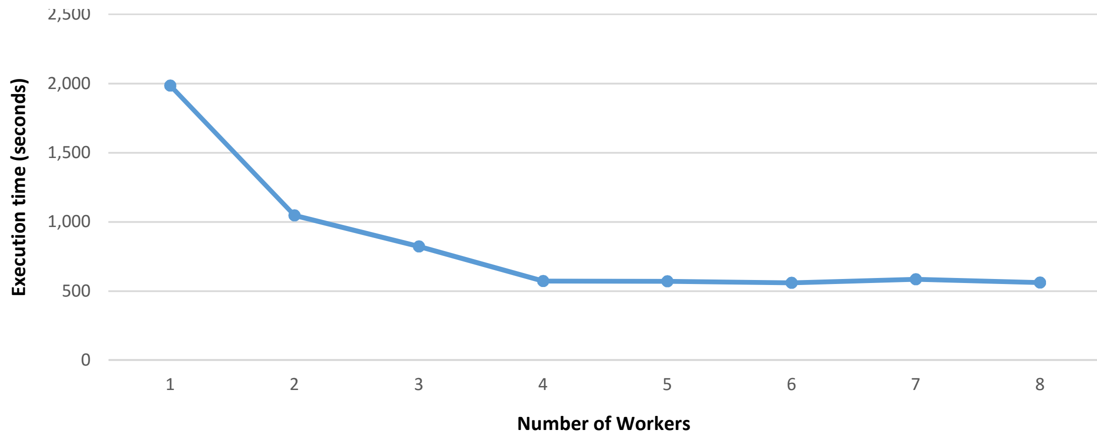


Figure 3.13: Execution time for an increasing number of workers.

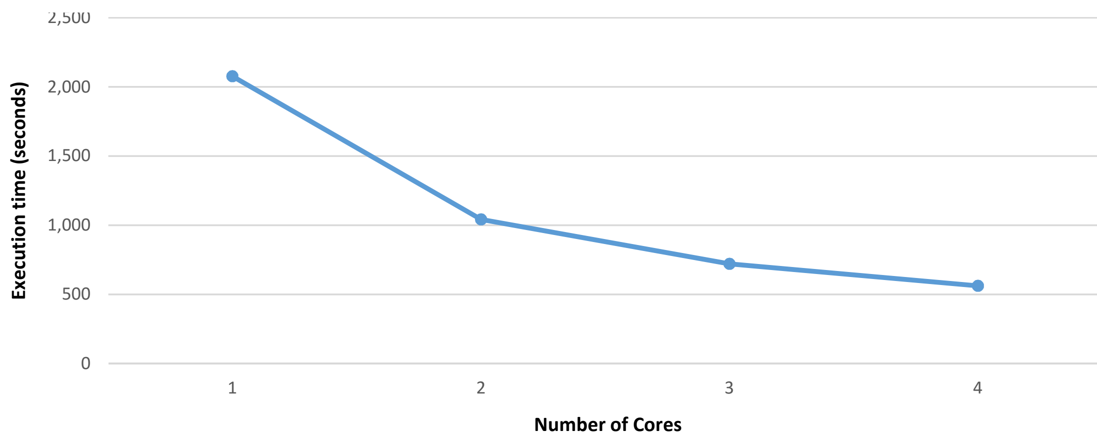


Figure 3.14: Execution time for an increasing number of cores.

3.3.4 Increasing number of cores

In this case, we change the number of cores of the virtual machine configuration. Fixing the configuration to 4 workers and modules, we increase the number of cores from 1 to 4. We have not used more cores because, in the computer used (see Section 3.3.1), the virtualization software drops its performance with 5 cores or more. The number of individuals per module is 2,500.

We can see in Figure 3.14 how our platform takes advantage of multi-core architectures. The computer with 4 cores runs 3.7 times faster than the one with one single core. The benefit is close to the maximum theoretical one (4 factors) [72].

3.3.5 Increasing number of modules and workers

In this experiment, we increase two variables at the same time. It is intended to represent a typical use case scenario. Assuming we have a multi-core computer (4 cores in our case), it is common to set the number of workers equal to the number of modules (or programs). The idea is trying to obtain the highest level of parallelization with a given computer. Therefore, we increase the number of modules and workers from 1 to 16. The number of functions is always 10,000,

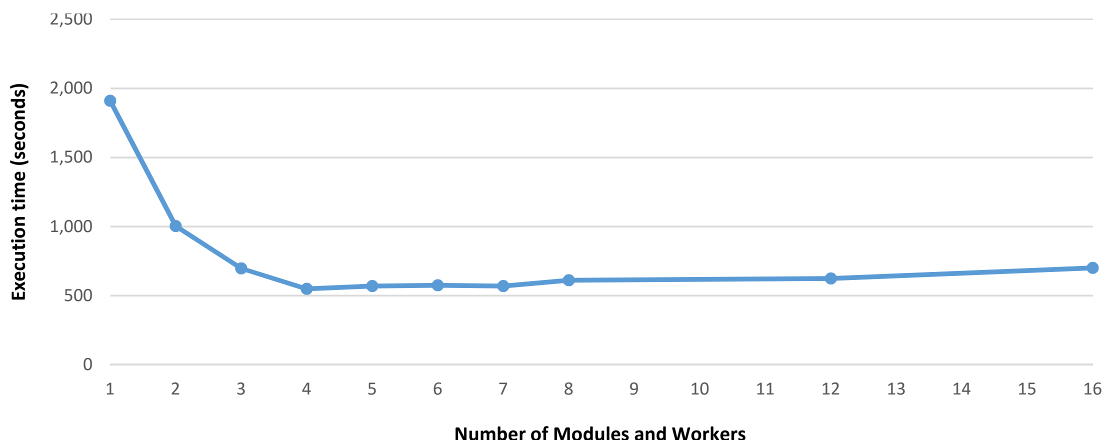


Figure 3.15: Execution time for an increasing number of modules and workers.

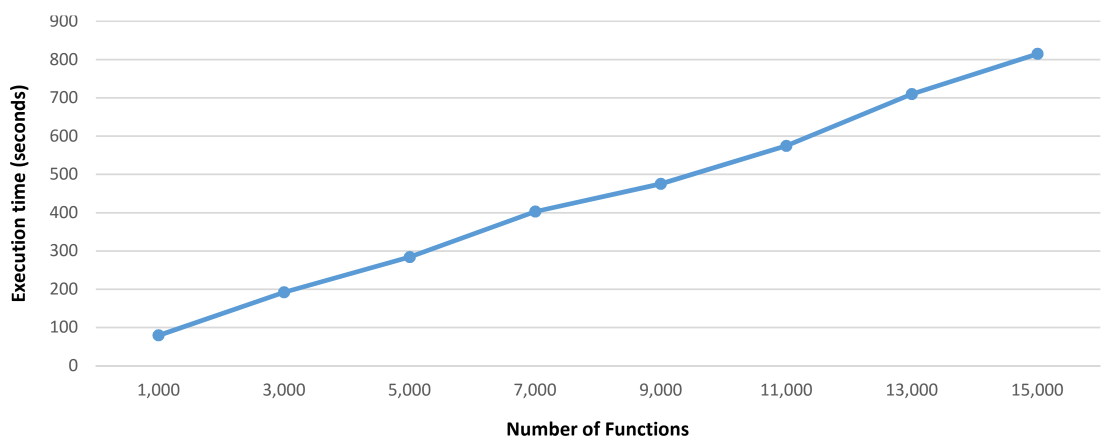


Figure 3.16: Execution time for an increasing number of functions.

equally distributed over the different modules of the program.

Figure 3.15 shows how execution time keeps reducing until 4 modules and workers (3.5 factors of benefit). From 4 to 7, differences among the values are lower than 1%. From 8 workers and modules on, the figure displays a slight increase in execution time due to the cost of context switching. Therefore, the results of the experiments seem to indicate that the optimal value for workers and modules ranges from the number of cores to twice this value.

3.3.6 Increasing number of functions

In order to see how the platform behaves for growing program sizes, this experiment increases the number of functions in the program from 1,000 to 15,000. We selected this maximum value because it was the biggest program supported by the IDA disassembler. The number of cores and workers is fixed to 4.

In Figure 3.16, we can see how execution time shows a linear increase of runtime performance for an increasing number of functions (*i.e.*, the size of the programs). The runtime performance of our platform is not spoiled when really big modules with 15,000 functions are processed.

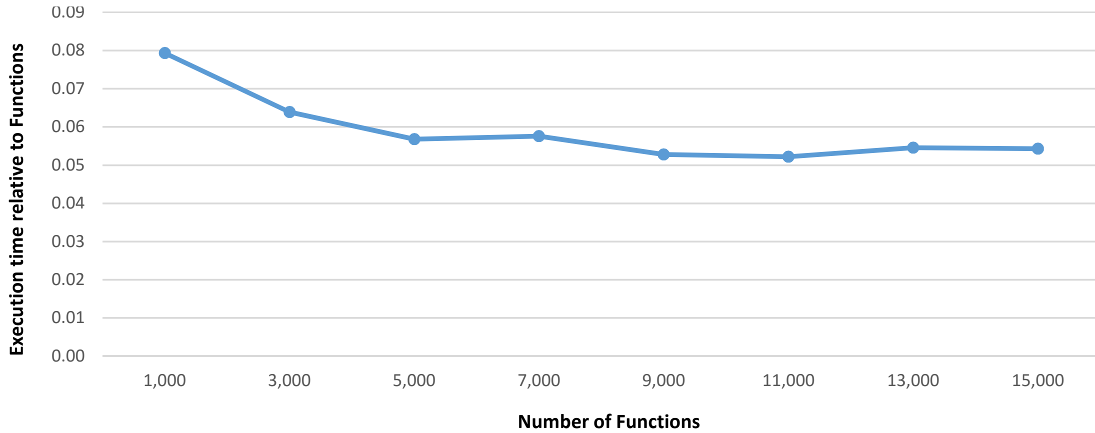


Figure 3.17: Execution time per function, increasing the number of functions.

Figure 3.17 presents another view of the same experiment. This figure displays the execution time performance per function, for an increasing number of functions in the program. For small programs, there is an initialization penalty that causes a higher execution time, when a low number of functions is processed. When the program size grows, this initialization cost becomes negligible. From 5,000 functions on, the execution time per function converges (the standard deviation is lower than 3.4%), showing that the runtime performance of the platform does not decrease for big input programs.

3.3.7 Increasing *MaxOffset* and *MaxSize*

We now modify the values of the *MaxOffset* and *MaxSize* parameters used to obtain the binary subpatterns. We used 4 modules, each one implemented with 750 functions. *MaxOffset* is incremented from 0 to 8, fixing *MaxSize* in 4. We apply the same method to analyze the influence of *MaxSize* on runtime performance, increasing its value from 1 to 8 and fixing *MaxOffset* to 4.

Figure 3.18 shows both variables. We can see how *MaxOffset* shows a linear influence on execution time. The regression line shown in Figure 3.18 has a slope of 51, representing the cost in seconds of increasing one unit in *MaxOffset* for the given configuration. For *MaxSize*, the best regression obtained is quadratic (Figure 3.18). The user should be aware of that, meaning that choosing high values for *MaxSize* will involve quadratic increases of the execution times.

3.3.8 Increasing types of patterns

The last variable to be measured is the number of patterns to be recognized. The patterns are specified with *Pattern Detection* functions provided by the user. In our return function type example, we identify 3 types of patterns (RET, PRE and POST CALL). We measure the runtime performance of the 7 possible combinations of these 3 patterns. Modules, workers, cores, *MaxSize* and *MaxOffset* are fixed to 4, and each module contains 750 functions (3,000 in total).

Figure 3.19 shows the results. The three first bars show the execution time consumed to extract each pattern individually. The three next bars display the

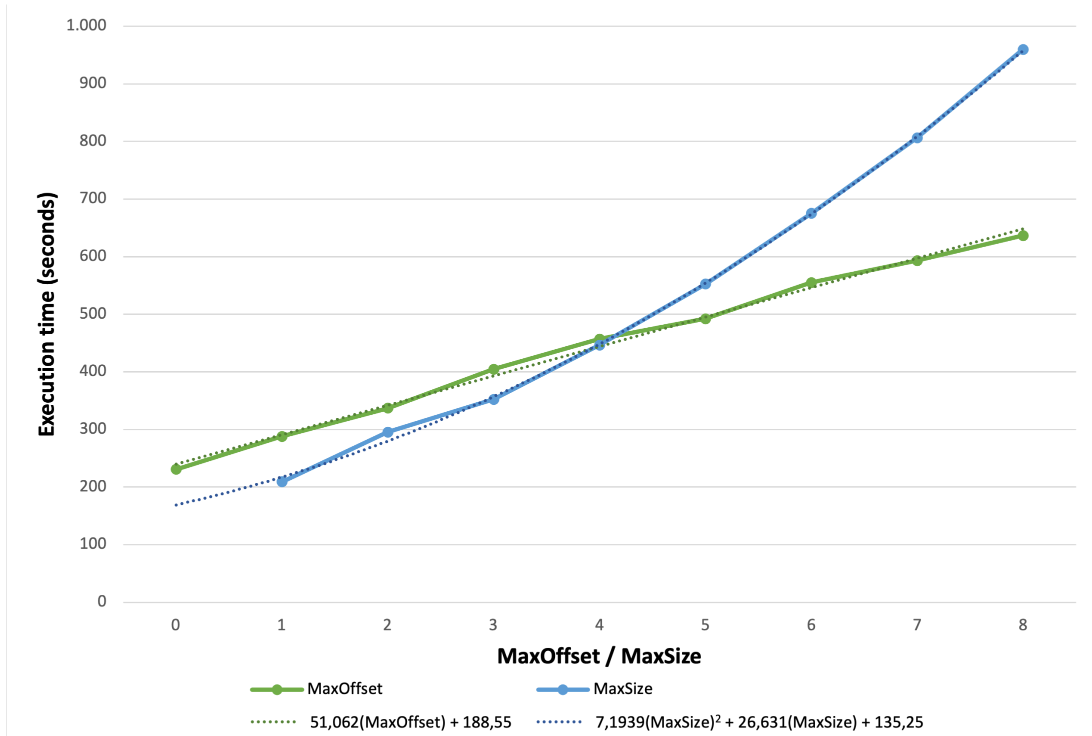


Figure 3.18: Execution time for an increasing number of *MaxOffset* and *MaxSize*.

execution time for two patterns in parallel, compared to the costs of extracting them individually. In these three scenarios, the platform obtains an average benefit of 1.65 factors due to parallelization. When the platform extracts three patterns in parallel, this benefit increases to 2.1 factors.

3.3.9 Execution time for a real case scenario

We have also measured execution time for the particular scenario of inferring the return type of a function. The purpose of this section is not to present how this problem can be solved with machine learning (detailed in Chapter 5), but to measure the execution time required to extract the binary patterns for that particular problem.

We extract binary code patterns before `ret` instructions, and before and after function invocations. Since a great number of functions are required to build an accurate model for this problem, we use the Cnerator stochastic source-code program generation tool (Chapter 7). With Cnerator, we can generate any number of random functions (and invocations to them) for all the different types in the language. These functions are then passed to our platform to generate the output dataset.

Since Cnerator provides us with any number of functions, we must work out the number of functions necessary to build an accurate model. For this purpose, we used the following method: we create 1000 functions for each C type; we extract the binary patterns in those functions with our platform; and we train a decision tree with the generated dataset to compute the accuracy rate using

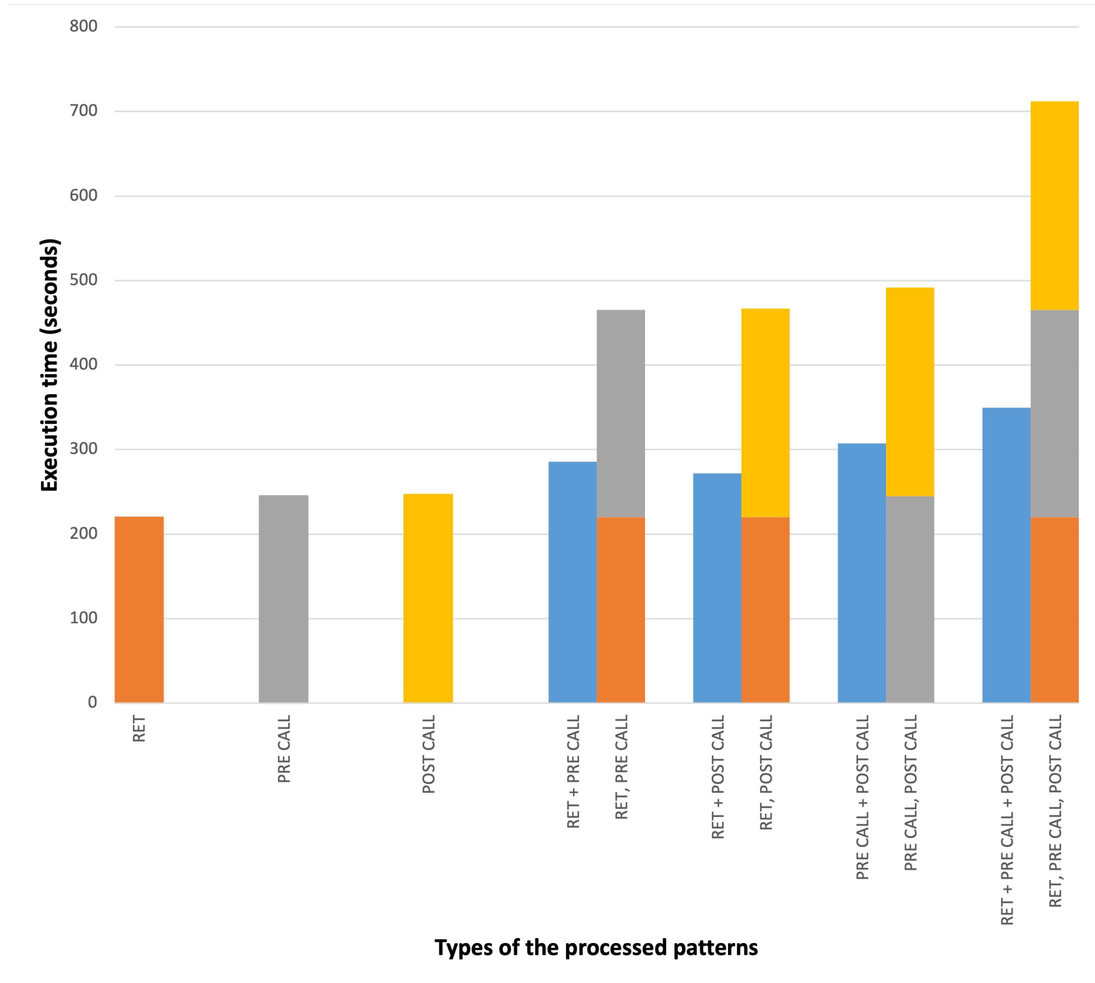


Figure 3.19: Execution time when extracting different types of patterns.

10-fold stratified cross-validation. These steps are repeated in a loop, incrementing the number of functions in 1000 for each type. We stop when the Coefficient of Variation (CoV) of the last 5 accuracy values is lower than 2%, representing that the increase of functions (individuals) does not represent a significant improvement of the accuracy. Finally, we build the final model with the dataset generated in the last iteration.

Following the method described above, we ended up with a dataset with 160,000 functions and 3,321 binary patterns (the size of the dataset was 998 MB). The platform generated the dataset in 2 hours 11 minutes and 56 seconds (4 workers and CPUs). We also measured the sequential version, taking 7 hours 41 minutes and 46 seconds to generate the same dataset.

The implementation of the platform is freely available for download [73].

Chapter 4

Method to create predictive models for decompilation

In the previous chapter, we described a platform for the automatic extraction of patterns in binary code. Before using this platform for a particular example of the decompilation problem, we propose a method to create predictive models aimed at improving the performance of existing decompilers. This is the general approach we propose to obtain high-level semantic information from binary code, using machine learning. As we will see, machine learning is not only used to build predictive models but also as a feature engineering mechanism to improve the datasets utilized to create such models.

4.1 Challenges faced

As mentioned, inferring the original source code from its compiled binaries is an undecidable problem [1]. Moreover, the creation of predictive models to improve existing decompilers involves many challenges, since such tools have been improved throughout the years [23]. A method to create predictive models for decompilation must tackle the following main challenges:

1. The first main challenge (Section 4.2) is the high dependency that the predictive models have on different variables, such as the compiler used, the target operating system and microprocessor and the compiler options. If the value of one of these variables is modified (*e.g.*, a different compiler is used), the binary patterns to be found in the compiled application change, and a new predictive model should be created to classify those patterns.

2. The second main challenge (Section 4.3) is that, for a given configuration (*i.e.*, set of values of the previously mentioned variables), the variability of binary code, and hence the search space, is huge. For example, the `mov eax, 5` assembly instruction is represented in x86 Windows 32-bit binary with the following five bytes: `b8 05 00 00 00`. Since the second parameter is a 4-byte `int` literal, the `mov eax` instruction may have $2^{32} = 4.294$ million distinct instances. This value is much higher than, for instance, the number

of words to be considered in a classical Natural Language Processing (NLP) problem (the English Wiktionary contains 520,000 entries).

3. The third main challenge (Section 4.4) is related to the language features to be decompiled. As we will see in this dissertation, inferring the original code of high-level language constructs (*e.g.*, the type returned by a function, its number of arguments, and control-flow statements) embodies sufficient complexity to build a separate model for each language construct.

4.2 Influencing variable

Different values of the following variables commonly produce distinct binary code when a source program is compiled:

- Compiler. The code generation module in a compiler takes the intermediate representation of a program and produces the output (binary) code [74]. The templates used to generate the code for a particular language construct vary from one compiler to another [75]. Therefore, the binary patterns to be classified depend on the compiler used to produce the binaries.
- Binary file format. Binary executables are those files that could be directly executed by the CPU. In addition to the binary representation of instructions, they contain headers and tables with relocation and fixup information, together with various types of metadata. Although there are plenty of binary file formats, the most common ones are Portable Executable (PE) for Windows, Executable and Linkable Format (ELF) for many versions of Unix, Mach-O for Mac OS and iOS, and MZ for DOS.
- Operating system. The binary code generated by a compiler commonly depends on the target operating system.
- Word/pointer size. Although most microprocessors currently provide a 64-bit word/pointer size, there are still many applications and operating system distributions for 32 bits. This fact should be considered when building predictive models to decompile binary code.
- Compiler options. Native language compilers (*e.g.*, C, C++, Go and Rust) provide multiple options for different purposes. The code optimization options and those aimed at specifying specific features of the generated code (*e.g.*, memory alignment and stack frame runtime error checking) influence the code templates used by the compiler. On the contrary, those options related to the compiler front-end (*e.g.*, type system, warning messages and semantic rules) should not be considered.
- Target microprocessor. There exist plenty of microprocessors in the market, and some binary language compilers provide specific code templates for each one. Although this could be considered as a particular compiler option (previous item), its strong influence on the generated code has made us include it as a different variable.

When one of the values of the previous variables changes, the patterns to be

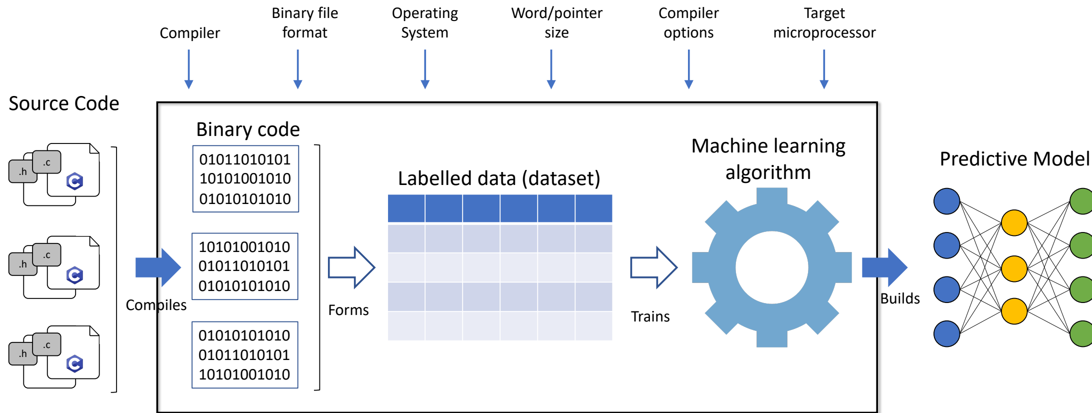


Figure 4.1: Variables that influence the decompilation predictive model.

found for the same high-level construct also change. This means that the decompilation problem strongly depends on those variables. Therefore, the method we propose to deal with this complexity creates different predictive models for each value of the given parameters (*i.e.*, each configuration).

Our proposal is depicted in Figure 4.1. Many high-level source code programs are compiled with a particular compiler, options, binary file format, operating system, word size and target microprocessor. Thanks to the instrumentation process described in Section 3.1.1, different binary patterns are labeled with high-level language constructs of the source code to build the dataset. A supervised machine learning algorithm is then used to train a predictive model, which is finally produced.

This process should be repeated for each value of the influencing variables (for each configuration) we want our system to be able to support. As a result, different predictive models are generated. Therefore, given a binary file to decompile, a relevant question would be which predictive model should be used to decompile that file.

Some values of the influencing variables may be inferred deterministically from the binary code. These variables are the binary file format, target microprocessor and word size [76]. For the rest of the variables, there exist probabilistic models capable of accurately inferring the compiler used [36], the compiler options [37] and the target operating system [77].

If we want to decompile binary code from different configurations, we should consider that training different models would require many CPU resources. However, once the predictive models are built, the runtime resources they consume are commonly low[1], and their implementation is pretty simple [9].

4.3 Variability of binary files

As mentioned, the binary language supported by existing microprocessors has a huge variability. Compared to word-level language models in NLP, the number

[1]Except for neighborhood-based algorithms such as k-nearest neighbors.

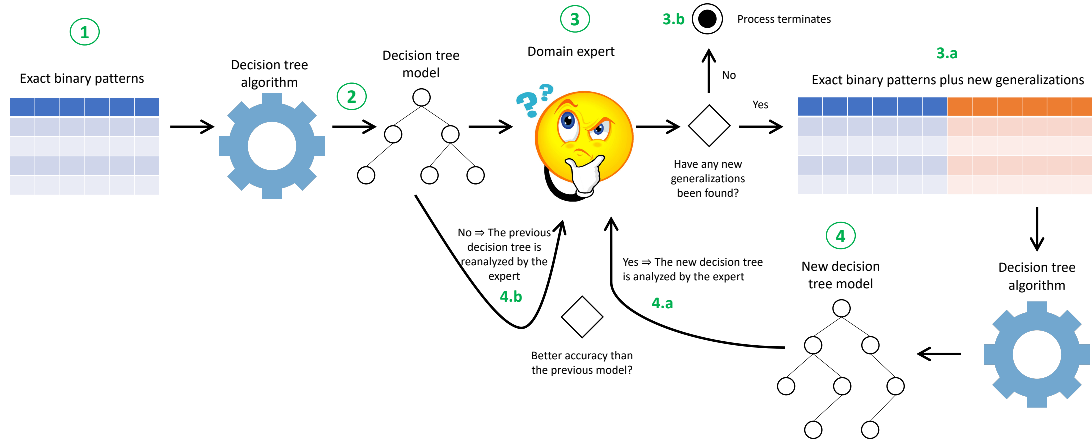


Figure 4.2: Feature engineering process.

of words used in such problems (around half a million) is much fewer than the raw representation of binary instructions (just the `mov eax,` *literal* instruction has more than 4.294 million distinct instances). Therefore, the automatic computation of word (byte) embeddings [78] for the decompilation problem does not seem to be feasible, due to the extremely huge number of programs that would be required in the training process –for example, English corpora with 1 million words are considered small [79].

Therefore, the exact representation of binary instructions cannot be directly used as features in the dataset, because we will not have sufficient programs (individuals, instances or samples) to train the models. Thus, the approach we propose is the generalization of binary constructs, supervised by a domain expert[1]. For example, most of the `mov eax,` *literal* instructions could be generalized to the same instance, when `int` is the high-level type returned by the function. However, `mov eax, 0` and `mov eax, 1` should not be included in that generalization, because they are often associated with the `bool` return type (see Chapter 6).

The generalization process constitutes a feature engineering mechanism to discover new and more general binary patterns (features in the dataset) that improve model accuracy. The discovery of such generalizations is done by a domain expert who analyzes the interpretable classification models created by a machine learning algorithm from a dataset without generalizations.

What follows are the steps of the iterative method we follow to define new binary pattern generalizations for a given syntax construct (*e.g.*, inferring the high-level type returned by a function), illustrated in Figure 4.2:

1. A dataset with the exact representation of binary instructions is created using the platform described in Chapter 3. For example, we include the RET, PRE and POST CALL binary patterns for each function, labeled with the actual high-level return type.

2. An interpretable white-box classifier is created and its classification perfor-

[1] In this context, domain experts are people with experience in compiler construction and a good understanding of assembly code.

mance is measured. We use decision trees because models are interpretable, perform well with large datasets, and handle both numerical and categorical data [80]. However, any other interpretable classification algorithm can be used.

3. The classification rules in the decision tree model are extracted. Then, the domain expert analyzes classification rules and the binary patterns found by the classifier. He/she checks the reason why the model successes or fails in the classification. For example, it could be discovered that the `mov eax`, *literal* pattern is classified as `int` except when the integer literal is `0` or `1`, which should be classified as `bool`.

   (a) If new generalizations are discovered, they are included in the dataset and the values of the cells for those new generalizations are computed. In our example, a new generalization pattern `mov eax`, *int literal excluding 0 and 1* is added to the dataset.

   (b) If no further generalizations are discovered, the algorithm terminates.

4. A new decision tree with the whole dataset, including the new features, is created. We compare its performance with one of the previous models.

   (a) If the performance of the new classifier is higher than the last one, then generalizations were successful and we keep the new features in the dataset. Jump to step 3.

   (b) If the classification performance decreases, the last generalization features are not kept in the dataset. Then, the domain expert should analyze why this occurs and propose new generalizations. Jump to step 3.

Table 4.1 shows a selection of some generalizations identified with our method for the decompilation of function return types (all of the generalizations are detailed in Appendix B). *Operand* is the most basic generalization type, which groups some types of operands, such as literals, addresses and indirections, into the same feature. *Mnemonic* generalizations group instructions with similar functionalities, such as `mov`, `movzx` and `movsx`. The last type of generalizations, *Sequence*, clusters sequences of instructions that appear multiple times in the binary code. For instance, *callee_epilogue* and *caller_epilogue* are binary sequences that appear, respectively, before returning one expression and after invoking a function.

We have seen how machine learning can be used not only to build predictive models but also as part of a feature engineering process (together with a domain expert) to improve the performance of the classifiers. For the particular case scenario of inferring the high-level return type of functions, the exact representation of binary instructions only provided 14% accuracy. With the generalizations included following our proposed method, 78.6% accuracy is achieved (see Chapter 5).

	Instruction sequences	Generalized pattern
Operand	sub al, 1	sub al, *literal*
	mov ecx, [ebp+var_1AC8]	mov ecx, [ebp+*literal*]
	mov ecx, [ebp+var_1AC8]	mov ecx, [*reg*]
	push offset $SG25215	push *address*
	movsd xmm0, ds:__real@43e2eb565391bf9e	movsd xmm0, **address*
	jmp loc_22F	jmp *offset*
	mov cx, [ebp+eax*2+var_10]	mov cx, [ebp+eax*$\mathit{literal}_1$+$\mathit{literal}_2$]
	mov cx, [ebp+eax*2+var_10]	mov cx, [ebp+*reg**$\mathit{literal}_1$+$\mathit{literal}_2$]
	mov cx, [ebp+eax*2+var_10]	mov ecx, [*reg*]
Mnem.	movzx ecx, [ebp+var_A]	*mov* ecx, [ebp+var_A]
	movsx ecx, _global_var_1234	*mov* ecx, _global_var_1234
	mov [eax], edx	*mov* [eax], edx
Sequence	pop esi pop edi mov esp, ebp pop ebp ret	*callee_epilogue*
	mov esp, ebp pop ebp ret	*callee_epilogue*
	pop ebp ret	*callee_epilogue*
	call _func56 add esp, 4	*caller_epilogue*
	call _proc2	*caller_epilogue*

	mov eax, 0
	mov ebx, eax
	mov [ebp+var_8], ebx
		mov_chain([ebp+var_8], ebx, eax, 0)

	ja loc_D9B6B
	mov [ebp+var_10], 1
	jmp loc_D9B72
	mov [ebp+var_10], 0
		bool_cast([ebp+var_10])

Table 4.1: Generalization examples made by our system. *reg* variables represent registers, *literal* integer literals, *address* absolute addresses, **address* absolute addresses dereferencing and *offset* relative addresses.

4.4 Language constructs

When decompiling a high-level program from binary files, we must first select the source language to be decompiled. Although any language that produces binary code could be considered (*e.g.*, C, C++, Go and Rust), C is commonly the *lingua franca* used by most compilers [81]. Most programmers know this language, and its abstraction level allows representing any language construct in an understandable fashion. However, if the user wants to decompile binaries to another high-level language, the following method could also be applied.

To decompile a specific language construct, there are some binary fragments that should be analyzed, whereas others do not affect the language constructs at all. For instance, the return type of a function depends on the binary code associated with the `return` high-level statements and on function invocations. However, the binary code that allocates memory for local variables does not affect the function return type. Thus, there is a strong dependency between the language construct to be inferred and some particular binary fragments.

In our study of the related work (Chapter 2) we saw how some predictive methods are more efficient to reconstruct particular language constructs. Chua *et al.* show how Recurrent Neural Networks (RNN) provide an accurate approach to infer the number of parameters [10]; extremely randomized trees combined with conditional random fields are used to discover names and types of identifiers; and RNN with additional post-processing techniques discover some code snippets. These works seem to indicate that is worth using potentially different models to decompile distinct language constructs.

The following steps describe the process we propose to create a predictive model for each different language construct:

Input: a collection of the language constructs to decompile.

Output: a collection of the predictive models.

Foreach language construct to be decompiled **do**

- Ask the domain expert to identify all the binary fragments that might influence the construct to infer.
- Label all the binary fragments with the high-level language construct identified by the expert. Create a dataset where each feature represents the distinct binary patterns found for each fragment, using the binary pattern extraction platform defined in Chapter 3.
- With that dataset:
  * Apply different feature selection algorithms (*e.g.*, recursive feature elimination and select from model).
  * Build interpretable white-box models (*e.g.*, decision trees).
  * Use different algorithms to obtain classification rules with minimum support and confidence (*e.g.*, rule ensembles [82] and FP-Growth [83]).

- The domain expert analyzes the interpretable white-box models, the selected and discarded features, and the classification rules obtained. With this information, he/she must identify the binary fragments that actually influence the language. Such fragments are included as features, and the remaining ones are no longer considered.
- Create the predictive model with the selected features, following the incremental iterative process described in Section 4.3.

Return the different predictive models created.

This process was followed for the decompilation problem of the function return type, for Windows PE 32-bit binaries generated by Microsoft C compiler with the default compiler options (Chapter 5). Initially, the domain expert included all the function body and the binary code before and after function invocation. The process finally indicated that only 1) the binary code for the expressions after `return`, and 2) the following instruction after function invocation (and caller epilogue), actually influence on the return type. We are about to see in the following chapter how to build predictive models from those two binary patterns.

Chapter 5

Decompilation of function return types

In this chapter, we apply the method proposed in Chapter 4 to decompile the return type of all the functions in a binary program. The resulting predictive models are evaluated and compared with the state-of-the-art decompilers. The platform described in Chapter 3 is used to retrieve the binary patterns from binary code. We focus our research on Windows PE 32-bit binaries generated by Microsoft C compiler with the default compiler options. For other kinds of binaries, the method described in Section 4.2 should be followed.

5.1 System overview

Figure 5.1 shows the overall view of our system [84]. It receives the C source code of multiple programs as an input and generates different machine learning models. Each model is aimed at predicting the high-level type returned by the functions in a program, by just receiving its binary representation.

The C source code is processed to build the datasets used to create the models. This process is performed with the binary pattern extraction platform described in Chapter 3. In this description, we summarize how we used our platform for this particular case scenario.

C source code is instrumented with annotations in the input program to allow associating high-level constructs to their binary representation (Section 3.1.1). Then, the instrumented source code is compiled to obtain the binaries. A pattern extraction process analyzes the binaries, looking for the annotations. It collects the set of binary patterns related to each function invocation and `return` statement. Finally, the resulting dataset is created, where binary patterns are associated with the return type of each function.

Table 5.1 shows the simplified structure of datasets generated by our platform. Each row (individual, instance or sample) represents a function from the input C source code. Each column but the last one (feature or independent variable) represents a binary pattern found by the pattern extraction process. The first

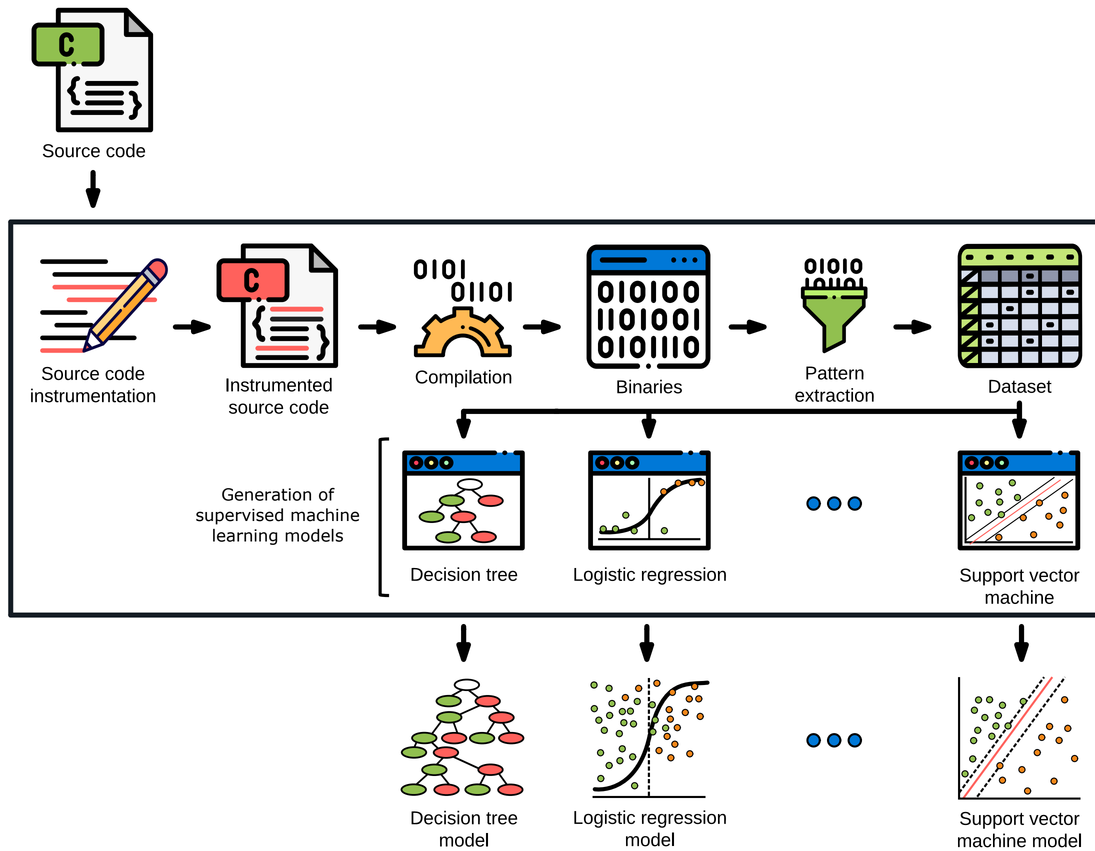


Figure 5.1: System architecture.

	(RET) `mov eax,` *literal* *callee_epilogue*	**(POST CALL)** *caller_epilogue* `cwde`	...	Return type
$func_1$	1	0	...	`int`
$func_2$	0	1	...	`short`
...	...	...	...	...
$func_{n\text{-}1}$	0	1	...	`short`
$func_n$	0	0	...	`double`

Table 5.1: Example dataset generated by our system.

```c
char toupper(char c) {
  if (c >= 'a' && c <= 'z')
    return c + 'A' - 'a';
  return c;
}
```

```c
char toupper(char c) {
  __dummy__("__04D__:toupper:char");
  if (c >= 'a' && c <= 'z')
    __RETURN1__: return c + 'A' - 'a';
  __RETURN2__: return c;
}
```

Figure 5.2: Original C source code (left) and its instrumented version (right).

pattern in Table 5.1 is the assembly code for a RET expression of some functions returning an `int` literal. That value is moved to the `eax` 32-bit register, followed by the code that all functions use to return to the caller (*callee_epilogue*). The second feature is the binary code of a function invocation (`caller_epilogue`) followed by a `cwde` instruction. Since `cwde` converts the signed integer representation from `ax` (16 bits) to `eax` (32 bits), the target class in the dataset is set to the `short` high-level type.

After creating the dataset, our system trains different classifiers following the methodology described in Section 4.2. The forthcoming subsections detail each of the modules of the system.

5.1.1 Instrumentation

As mentioned, much high-level information is discarded in the compilation process. One example is the association between a high-level `return` statement and its related assembler instructions. There is not a direct way to identify the binary code generated for a `return` statement. For this reason, our instrumentation process includes no-operational code around some syntactic constructs in the input C program. The instrumented code does not change the semantics of the program but helps us find the binary code generated for different high-level code snippets

The left-hand side of Figure 5.2 shows an example of the original C function, and its right-hand side presents the instrumented version. The function `__dummy__` performs no action. Its invocation is added to provide information about the name and return type of the high-level function in the binary code.

The example in Figure 5.2 also shows the instrumentation to delimit the binary code of the expressions returned by a function. To this end, different `__RETURN`*n*`__` labels are added before every `return` statement. The end of the returned expression is delimited by the `ret` assembly instruction.

5.1.2 Compilation

The instrumented C source code is compiled to obtain the binaries. In this case, we only use the Microsoft `cl` compiler to build native applications for Intel x86 32-bit microprocessors. The compilation parameters are the default ones. The method described in Section 4.2 should be followed if any of these variables (compiler, file format, operating system, word size, compiler options or target

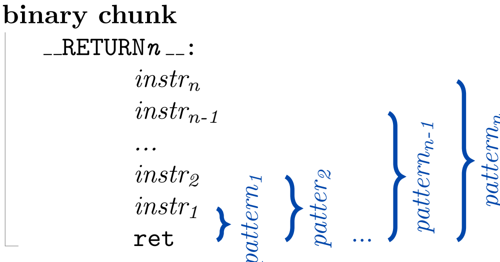


Figure 5.3: A RET binary fragment/chunk

microprocessor) are changed.

5.1.3 Pattern extraction

The pattern extraction module performs three processes: extraction of binary fragments or chunks, pattern generalization and dataset generation.

5.1.3.1 Extraction of binary fragments

The first task is to extract the binary fragments or chunks associated with every `return` and function invocation statements. As we saw in Section 3.1.2, we can use the Binary Pattern Extractor module of our platform to retrieve RET and POST CALL patterns. Then, with the method described in Section 4.4, we found out that PRE CALL patterns have almost no influence on the type returned by a function[1], so we do not include them in the output dataset.

For the RET patterns, we extract the binary chunks between `__RETURN`*n*`__` labels and the next `ret` instruction. Since we do not know how many instructions are sufficient to predict the return type (one high-level expression may produce many low-level instructions), we generate different binary patterns with a growing number of binary instructions before `ret`. To do that, we set *MaxOffset* to 0 and *MaxSize* to the number of binary instructions between `ret` and the `__RETURN`*n*`__` labels (Figure 5.3).

For the POST CALL patterns, we retrieve the sequences of instructions after each function invocation. We consider the `call` instruction, the optional `add esp,` *literal* instruction used to pop the invocation arguments from the stack, and the following assembly instructions. Similar to RET patterns, we first created different POST CALL patterns with an increasing number of instructions after `call` and stack restoration. However, the application of the method in Section 4.4 showed us that only the first assembly instruction after stack restoration influences on the returned type. Consequently, this was the only instruction considered in POST CALL patterns ($MaxOffset = 1$ and $MaxSize = 2$).

[1] As we will see in Section 5.3.1, the PRE CALL patterns are only useful to detect struct types, due to the code transformation that the Microsoft `cl` compiler implements to return structs (Figure 5.4). However, as we will see in Chapter 6, RET and POST CALL patterns are sufficient to detect structs.

Project	Functions	LoC	Description
arcadia	121	3,590	Implementation of Arc, a Lisp dialect.[a]
bgrep	5	252	Grep for binary code.[b]
c_ray_tracer	52	1,063	Simple ray tracer.[c]
jansson	176	7,020	Library for encoding, decoding and manipulating JSON data .[d]
libsodium	642	35,645	Library for encryption, decryption, signatures and password hashing[e].
lua 5.2.3	820	14,588	The Lua programming language.[f]
masscan	496	26,316	IP port scanner.[g]
slre	17	564	Regular expression library.[h]
Total	2,329	89,038	

[a] https://github.com/kimtg/arcadia
[b] https://github.com/elektrischermoench/bgrep
[c] https://web.archive.org/web/20150110171135/http://patrickomatic.com/c-ray-tracer
[d] https://github.com/akheron/jansson
[e] https://github.com/jedisct1/libsodium
[f] https://lua.org/download.html
[g] https://github.com/robertdavidgraham/masscan
[h] https://github.com/cesanta/slre

Table 5.2: Open-source C projects used.

5.1.3.2 Pattern generalization

In Section 4.3, we discussed the huge variability of binary files. That variability demands a generalization mechanism to use machine learning for decompilation purposes. The algorithm described in Figure 4.2 is applied to generalize the RET and POST CALL patterns selected by our platform. Those generalizations improve the performance of the predictive models. All the generalizations discovered for this particular decompilation problem are detailed in Appendix B.

5.1.3.3 Dataset creation

After pattern generalization, the datasets are created before training the models (Table 5.1). Our platform gets the return type of each function (*i.e.*, the target or dependent variable) from the string parameter passed to the `__dummy__` function added in the instrumentation process (Section 5.1.1).

5.2 Methodology

We describe the methodology used to build and evaluate the predictive models to infer the high-level types returned by functions.

5.2.1 Data origin

We take different C programs from open-source repositories to build our models. Table 5.2 shows the different open-source C projects taken to create the dataset. Although they sum 2329 functions and 89,038 lines of code, they do not represent sufficient data to infer return types. Unfortunately, there are not many open-source C (not C++) programs compilable with Microsoft `cl` compiler. Moreover,

to balance the dataset in Table 5.2, we may need to ignore some functions, so the final number of functions would even be lower. Thus, we use Cnerator, our stochastic C source code generator tool (Chapter 7), to increase the size of our dataset.

The final dataset comprises the source code of the "real" projects in Table 5.2 plus the synthetic code generated by Cnerator. On one hand, the synthetic code provides a huge number of functions, a balanced dataset, and all the language constructs we want to include. On the other hand, real projects increase the probability of those patterns that real programmers often use (*e.g.*, most C programmers use `int` expressions instead of `bool` for Boolean operations). This combination of real and synthetic source code improves the predictive capability of our dataset.

5.2.2 Data size

Since Cnerator allows us to generate any number of functions (*i.e.*, individuals), we should define the appropriate number of functions to create our dataset. To determine this number, we conduct the following experiment. We start with a balanced dataset with 100 functions for each return type. Then, we build different classifiers (see Section 5.2.3) and evaluate their accuracy. Next, we add 1000 more synthesized functions to the dataset, re-build the classifiers and re-evaluate them. This process is repeated until the accuracy of classifiers converge. To detect this convergence, we compute the coefficient of variation (CoV) of the last ten accuracies, stopping when that coefficient is lower than 2%.

5.2.3 Classification

We use the following 14 classifiers from scikit-learn [85]: logistic regression (`LogisticRegression`), perceptron (`Perceptron`), multilayer perceptron (`MLPClassifier`), Bernoulli naïve Bayes (`BernoulliNB`), Gaussian naïve Bayes (`GaussianNB`), multinomial naïve Bayes (`MultinomialNB`), decision tree (`DecisionTreeClassifier`), random forest (`RandomForestClassifier`), extremely randomized trees (`ExtraTreesClassifier`), support vector machine (`SVC`), linear support vector machine (`LinearSVC`), AdaBoost (`AdaBoostClassifier`), gradient boosting (`GradientBoostingClassifier`), k-nearest neighbors (`KNeighborsClassifier`).

In the process described in Section 5.2.2 to find the optimal size of the dataset, we use a stratified and randomized division of the dataset (`StratifiedShuffleSplit` class in scikit-learn). 80% of the instances in the dataset are used for training and the remaining 20% for testing. Since each classifier has a different optimal size, we choose the greatest optimal size (results are shown in Sections 5.3.1.1 and 5.3.2).

5.2.4 Feature selection

Our system generates a lot of features because, for each pattern, different generalizations are produced. Therefore, a feature selection mechanism would be

beneficial to avoid the curse of dimensionality and enhance the generalization property of the classifiers. Therefore, after creating the datasets with the optimal size, we select the appropriate features to create each model.

We use recursive feature elimination (`RFECV`) and selection from a model (`SelectFromModel`) wrapper methods [86]. Given an external estimator that assigns weights to features, `RFECV` selects the features by recursively considering smaller sets. The feature selection process has been applied to the training dataset (80% of the original one) using 3-fold stratified cross-validation (`StratifiedShuffleSplit`).

`RFECV` can only be used with algorithms that use scores. For the rest of the algorithms (naïve Bayes, k-nearest neighbors, multilayer perceptron and AdaBoost), we utilized the `SelectFromModel` feature selection meta-transformer. `SelectFromModel` discards the features that have been rejected by some other classifiers like tree-based ones. We used four `SelectFromModel` configurations: random forest and extremely randomized trees as classifiers to select the features; and the mean and median thresholds to filter features by their importance score. The feature set with the best accuracy is selected for each different classification algorithm (see the results in Sections 5.3.1.2 and 5.3.2).

5.2.5 Hyperparameter tuning

After feature selection, we tune the hyperparameters of each model. To that end, we use `GridSearchCV`, which performs an exhaustive search over the specified hyperparameter values. Similar to the feature selection process, the 80% training set is used to validate the hyperparameters with 3-fold stratified cross-validation (`StratifiedShuffleSplit`).

The final hyperparameters selected for each classifier are available at Appendix D. For the multilayer perceptron neural network, we use a single hidden layer with 100 units, the sigmoid activation function, Adam optimizer, and softmax as the output function.

5.2.6 Evaluation of model performance

After feature selection and hyperparameter tuning, we create and evaluate different models (one for each algorithm in Section 5.2.3) to predict the type returned by functions. As mentioned, the dataset has real functions coded by programmers, and synthetic ones generated by Cnerator. We consider these two types of code to define three different methods to evaluate the performance of the classifiers:

1. Mixing real and synthetic functions. This is the simplest evaluation method, where real and synthetic functions are merged in the dataset. 80% of them are used for training and the remaining 20% for testing. These two sets are created with stratified randomized selection. Therefore, real and synthesized functions in the training and test sets are, respectively, 80% and 20%.

2. Estimate the necessary number of real functions for training. Since we have lots of synthetic functions, we want to estimate to what extent synthetic programs can be used to classify code written by real programmers. We first create a model only with synthetic functions (80%) and test it with real ones (20%). Then, we include 1% of real functions in the training dataset, rebuild and retest the models, and see the accuracy gain. This process stops when the CoV is lower than 1% for the last 10 accuracies. The obtained percentage of real functions in the training set indicates how many real functions are necessary to build accurate predictive models (32% for the experiment in Section 5.3.1 and 48% for that in Section 5.3.2). Decision tree was the classifier used to estimate this value.

3. Prediction of complete real programs. This evaluation method measures prediction for source code written by programmers whose code has not been included in the test dataset. One real program is used to build the test dataset, and no functions of that program are used for training. In this case, we evaluate whether our system is able to predict return types for unknown programming styles.

5.2.7 Selected decompilers

We compare our models with the following existing decompilers:

- IDA Decompiler [46]. This is a plugin of the commercial Hex-Rays IDA disassembler [87]. This product is the result of the research works done by Ilfak Guilfanov [47, 48]. This tool is the current *de facto* standard in software reverse engineering.

- RetDec [55]. An open-source decompiler developed initially by Křoustek [56], currently maintained by the AVAST company. It can be used as a standalone application or as a Hex-Rays IDA plugin. To avoid the influence of the Hex-Rays IDA decompiler, we use the standalone version.

- Snowman [52]. Open-source decompiler based on the TyDec [50] and SmartDec [51] proposals. Similar to RetDec, it can also be used as a standalone application or as a Hex-Rays IDA plugin. We use the standalone version.

- Hopper [53]. A commercial decompiler developed by Cryptic Apps. Although it is mainly focused on decompiling Objective-C, it also provides C decompilation of any Intel x86 binary.

We also considered other alternatives that we finally did not include in our evaluation. DCC [2] and DISC [42] decompilers do not work with modern executables. The former is aimed at decompiling MS-DOS binaries, while the latter only decompiles binaries generated with TurboC. Boomerang [3] and REC [45] are no longer maintained. Lastly, we did not find the implementations of the Phoenix [54], DREAM [58] and DREAM++ [59] decompilers.

5.2.8 Data analysis

For each classifier, we compute its performance following the three different evaluation methods described in Section 5.2.6. We repeat the training plus testing process 30 times, computing the mean, standard deviation and 95% confidence intervals of model accuracies. This allows us to compare accuracies of different models, checking whether two evaluations are significantly different when their two 95% confidence intervals do not overlap [88]. Figures showing model accuracies (Figure 5.7 and Figure 5.10) display the 95% confidence intervals as whiskers.

In a balanced multi-class classification, overall precision and recall are usually computed as the average of the metrics calculated for each class. These aggregate metrics are called macro-precision and macro-recall [89]. Likewise, macro-F_1-score can be computed as the average of per-class F_1-score [89], or as the harmonic mean of macro-precision and macro-recall [90]. We use the first alternative because it is less sensitive to error type distribution [91]. For the sake of brevity, we use precision, recall and F_1-score to refer to the actual macro-precision, macro-recall and macro-F_1-score measurements.

We run all the code in a Dell PowerEdge R530 server with two Intel Xeon E5-2620 v4 2.1 GHz microprocessors (32 cores) with 128GB DDR4 2400 MHz RAM, running an updated version of Windows 10 for 64 bits.

5.3 Evaluation

In the assembly language, the concept of type is related to the size of values more than to the operations that can be done with those values. For example, the integer `add` instruction works with 8 (`ah`), 16 (`ax`) and 32 bits (`eax`), but it is not checked whether the accumulator register is actually holding an integer. For this reason, in this section, we evaluate two different kinds of models: those considering types by their size and representation (Section 5.3.1), and those considering types by the operations they support—*i.e.*, high-level C types (Section 5.3.2).

The first kind of models predicts return types when they have different size or representation. In this way, these models separate `short` (2 bytes) from `int` (4 bytes). They also tell the difference between `int` and `float` because, even though their size is 4 bytes, their representations are different (integer and real). On the contrary, `char` and `bool` are not distinguished in the first kind of models because they both are 1-byte sized and hold integer values (C does not provide different operations for `char` and `bool`).

After building and evaluating these type-by-size-and-representation models (Section 5.3.1), we define additional mechanisms to distinguish among types with similar sizes. Thus, Section 5.3.2 shows additional generalization patterns obtained with the method in Section 4.3 to improve the classification of high-level return types. With those enhancements, our models improve the differentiation among types with the same size such as `char` and `bool`, and `int` and `pointer`[1].

[1]Note that, in assembly, there is no difference in the *representation* of integers, characters, Booleans and pointers, because, for the microprocessor, all of them hold integer values.

Target	C high-level type
INT_1	bool and char
INT_2	short
INT_4	int, long, pointer, enum and struct
INT_8	long long
REAL_4	float
REAL_8	double and long double
VOID	void

Table 5.3: Relationship between the target variable (types grouped by size and representation) and the C high-level types.

```
typedef struct stats stats_t;

stats_t init_stats() {
  stats_t s;
  /* ... function code ... */
  return s;
}


void main() {
  stats_t s = init_stats();
  /* ... some code ... */
  double mean = ((double)s.sum) /
  ↪  ((double)s.count);
}
```

```
typedef struct stats stats_t;

stats_t * init_stats(stats_t *result)
↪  {
  stats_t s;
  /* ... function code ... */
  *result = s;
  return result;
}

void main() {
  stats_t s;
  stats_t *result = init_stats(&s);
  /* ... some code ... */
  double mean = ((double)result->sum)
  ↪  / ((double)result->count);
}
```

Figure 5.4: The left-side code is transformed by `cl` into the right-side code to allow returning structs in `eax`, which are actually passed to the caller as pointers.

5.3.1 Grouping types by size

We start with the evaluation of models that group types by their size and representation. We apply all the steps defined in the methodology (Section 5.2). Then, we repeat the same process for models considering the C high-level types (Section 5.3.2).

In binary code, the value returned by a function is passed to the caller via registers. CPUs have different kinds of registers depending on their sizes and representations (integer or a floating-point). In the particular case of Intel x86, registers can hold integer values of 8-, 16- or 32-bit, and 32- or 64-bit floating-point numbers.

Table 5.3 shows the target variable used in this first kind of models and their corresponding C type. For INT_2, INT_8, REAL_4 and VOID, the class used corresponds with a single high-level type. REAL_8 groups double and long double, while INT_4 considers all the C types returned in the 32-bit eax register (structs are actually returned as pointers).

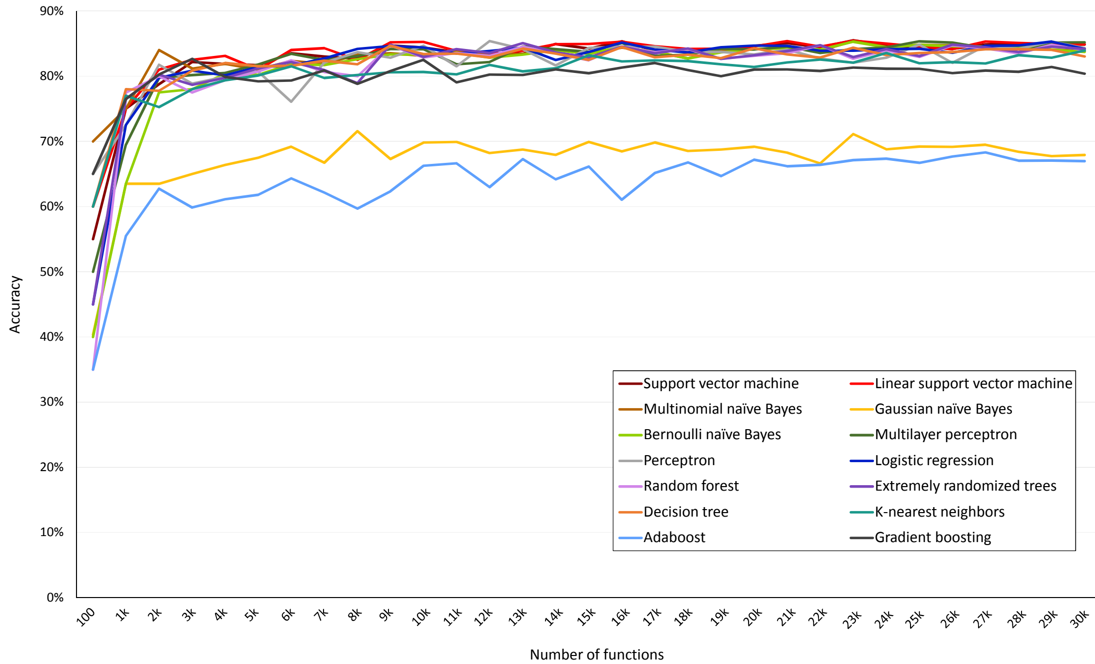


Figure 5.5: Classifiers accuracy for increasing number of functions (classifiers of types with different size and representation).

Pointers are represented as `INT_4` because the size of memory addresses in Intel x86 is 32 bits (4 bytes). The `struct` type is also clustered as `INT_4` because the `cl` compiler transforms returned structs into pointers to structs, as depicted in Figure 5.4. The returned pointer to `struct` is actually the `result` pointer passed as an argument. In this way, the actual `struct` is a local variable created in the scope of the caller (`s` variable in Figure 5.4), making easy the management of the memory allocated for the `struct`. This is the reason why the actual value returned is not a `struct` but a `pointer` (4 bytes).

The `union` type constructor is not listed in Table 5.3 because it has variable size and representation. When the size of the biggest field is not bigger than 32 bits, 4 bytes are used. When it is higher than 4 bytes and lower or equal to 8, 64 bits are used. In case it is greater than 8 bytes, the compiler generates the same code as for structs (Figure 5.4).

5.3.1.1 Data size

As mentioned in the methodology (Section 5.2), we use Cnerator to produce a dataset with such a number of functions that make models accuracies to converge. Figure 5.5 shows how classifiers accuracies grow as the dataset size increases. Figure 5.6 presents the CoV of the last 10 values. We can see how, with 26,000 functions, the CoVs of the accuracies for all the classifiers are below 2%. Since CoV is computed for the last 10 iterations and each iteration increases 1,000 functions, we build the dataset with 16,000 functions (and their invocations). In addition to those 16,000 instances, we add the 2,329 functions retrieved from real programs (Table 5.2).

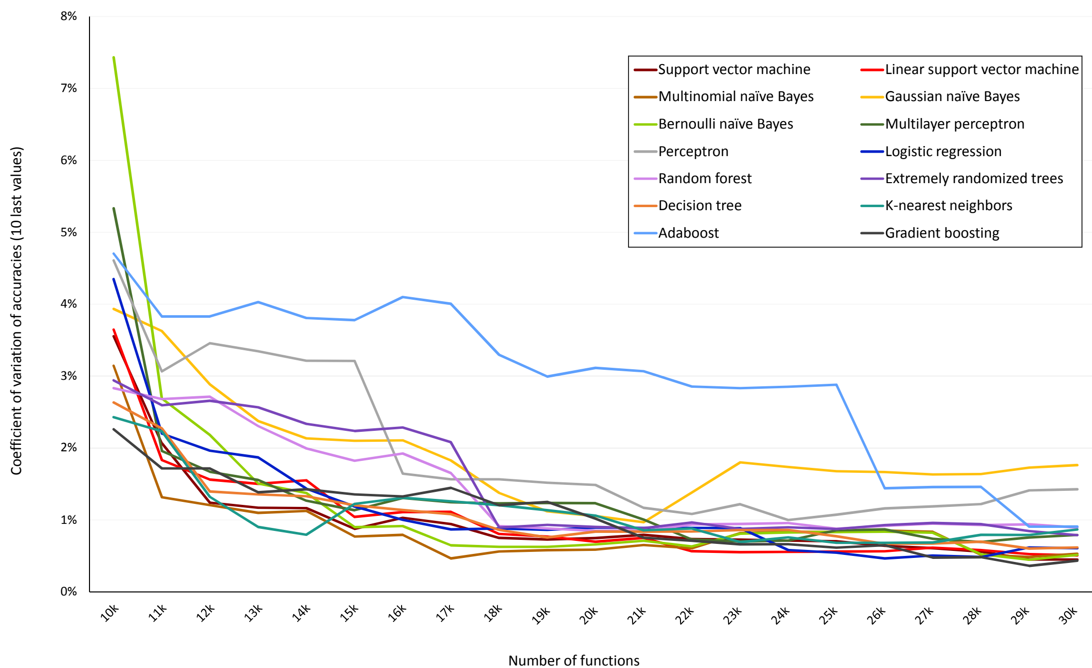


Figure 5.6: Coefficient of variation of the last 10 accuracy values in Figure 5.5 (classifiers of types with different size and representation).

5.3.1.2 Feature selection

We apply the five feature-selection methods described in Section 5.2.4 (`RFECV` and `SelectFromModel` with random forest and extremely randomized trees, with mean and median thresholds). Table 5.4 shows the best feature selection method for each classifier, together with the number of features selected. The 1,019 features of the original dataset are reduced, on average, to 366. The selected features produce statistically significant higher accuracy for Gaussian naïve Bayes (3.22% better) and multilayer perceptron (3.52%). Moreover, the lower number of features reduces training times and overfitting.

Classifier	Applied method (wrapped algorithm, accuracy threshold)	Selected features
AdaBoost	SelectFromModel(Random forest, Mean)	134
Bernoulli naïve Bayes	SelectFromModel(Random forest, Median)	439
Decision tree	SelectFromModel(Extremely randomized trees, Median)	419
Extr. randomized trees	SelectFromModel(Extremely randomized trees, Median)	419
Gaussian naïve Bayes	SelectFromModel(Extremely randomized trees, Mean)	130
Gradient boosting	SelectFromModel(Extremely randomized trees, Median)	419
K-nearest neighbors	SelectFromModel(Random forest, Median)	439
Lin. sup. vector machine	SelectFromModel(Random forest, Median)	439
Logistic regression	SelectFromModel(Random forest, Median)	439
Multilayer perceptron	SelectFromModel(Extremely randomized trees, Median)	419
Multinomial naïve Bayes	SelectFromModel(Random forest, Median)	439
Perceptron	SelectFromModel(Random forest, Median)	439
Random forest	SelectFromModel(Random forest, Mean)	134
Support vector machine	SelectFromModel(Extremely randomized trees, Median)	419

Table 5.4: Best feature selection method used for each classifier (classifiers of types with different size and representation).

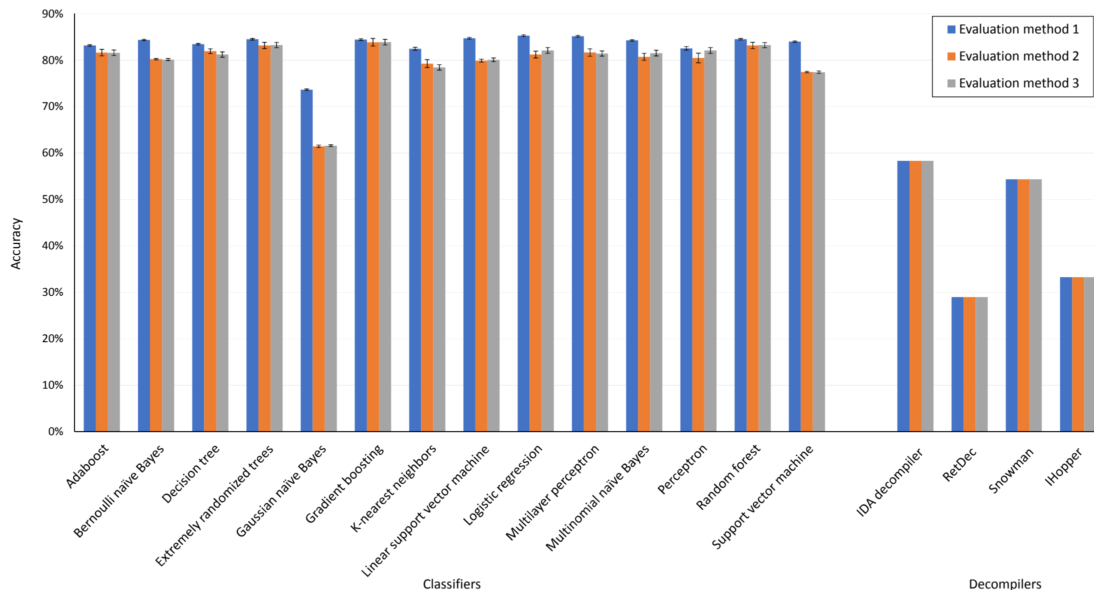


Figure 5.7: Accuracies of our classifiers and the existing decompilers, using the three different evaluation methods described in Section 5.2.6 (classifiers of types with different size and representation).

5.3.1.3 Hyperparameter tuning

We tune hyperparameters of the models as described in Section 5.2.5. For the hyperparameters found, AdaBoost increased its accuracy by 7.85%. However, the rest of the classifiers obtained accuracy gains below 2%, compared to the scikit-learn default parameters. The final hyperparameters selected for each classifier are available at Appendix D.

5.3.1.4 Results

Figure 5.7 shows the accuracies of the 14 trained models (left-hand side) and the selected decompilers (right-hand side). All the systems are evaluated with the three methods described in Section 5.2.6. It can be seen how all the classifiers created with our dataset perform better than the existing decompilers, for all the evaluation methods.

In Figure 5.7, we can also see that there are significant differences between the first evaluation method and the two last ones, for all the machine learning models. This shows how the common evaluation method that takes 80% for training and 20% for testing is too optimistic for this work. We need to feed the models with sufficient code written by real programmers so that we are able to predict return types with different programming styles. Existing decompilers show no influence on the evaluation method because they use deterministic algorithms to infer return types.

One discussion related to the second evaluation method is finding out the number of real functions to be included in the training set, so that return types could be inferred for unknown real code. As described in Section 5.2.6, we start with a test set of 100% synthesized functions, and incrementally add real functions

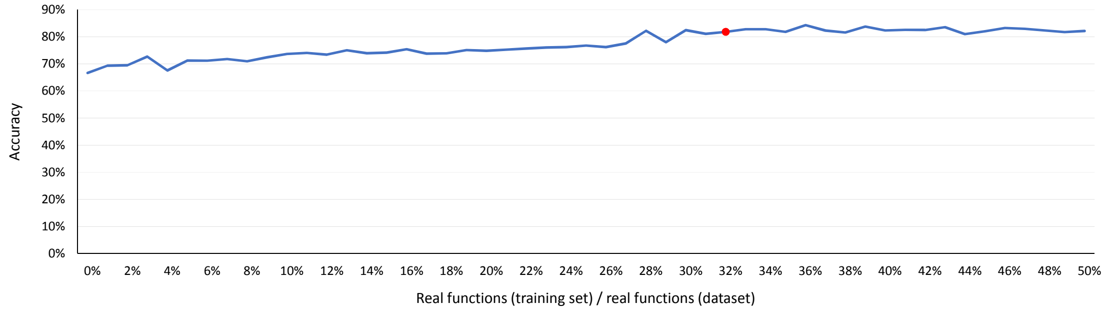


Figure 5.8: Accuracy of a decision tree for different percentage of real functions included in the training dataset (classifiers of types with different size and representation). The red dot indicates the value where the CoV of the last 10 accuracies is lower than 1%.

		Accuracy	Precision	Recall	F_1-score
Classifiers	AdaBoost	0.816 ± 1.40%	0.821 ± 1.10%	0.815 ± 0.54%	0.805 ± 1.01%
	Bernoulli naïve Bayes	0.801 ± 0.51%	0.801 ± 0.43%	0.830 ± 0.35%	0.798 ± 0.42%
	Decision tree	0.812 ± 1.45%	0.817 ± 1.21%	0.814 ± 0.55%	0.803 ± 1.06%
	Extr. randomized trees	**0.833** ± 1.39%	0.839 ± 1.22%	**0.831** ± 0.50%	0.823 ± 1.04%
	Gaussian naïve Bayes	0.616 ± 0.59%	0.687 ± 0.61%	0.736 ± 0.35%	0.650 ± 0.60%
	Gradient boosting	**0.839** ± 1.44%	**0.863** ± 1.15%	**0.832** ± 0.51%	**0.838** ± 1.05%
	K-nearest neighbors	0.785 ± 1.51%	0.798 ± 1.22%	0.805 ± 0.54%	0.783 ± 1.07%
	Lin. sup. vector machine	0.801 ± 0.95%	0.827 ± 1.32%	0.823 ± 0.43%	0.806 ± 0.90%
	Logistic regression	0.821 ± 1.52%	0.817 ± 1.37%	**0.834** ± 0.50%	0.812 ± 1.15%
	Multilayer perceptron	0.814 ± 1.37%	0.821 ± 1.24%	**0.831** ± 0.46%	0.809 ± 0.94%
	Multinomial naïve Bayes	0.815 ± 1.51%	0.819 ± 1.32%	0.829 ± 0.52%	0.807 ± 1.14%
	Perceptron	0.821 ± 1.53%	0.856 ± 1.74%	0.815 ± 0.82%	0.818 ± 1.28%
	Random forest	**0.833** ± 1.37%	0.840 ± 1.15%	**0.830** ± 0.49%	0.823 ± 1.00%
	Support vector machine	0.774 ± 0.64%	0.807 ± 1.35%	0.809 ± 0.37%	0.782 ± 0.70%
Decomp.	IDA decompiler	0.583	0.495	0.413	0.415
	RetDec	0.290	0.111	0.133	0.110
	Snowman	0.544	0.365	0.328	0.322
	Hopper	0.333	0.132	0.132	0.079

Table 5.5: Performance of the classifiers and existing decompilers using the third evaluation method (classifiers of types with different size and representation). 95% confidence intervals are expressed as percentages. Bold font represents the best values. If one column has multiple cells in bold, it means that values are not significantly different.

until model accuracy converges. Figure 5.8 shows this influence of real functions on the classifier accuracy. The red dot shows that with 32% of real functions, CoV of model accuracy falls below 1%. We fixed that value for the second evaluation method.

Figure 5.7 also shows that there is no statistically significant difference between the second method and the third one (*i.e.*, 95% confidence intervals overlap [88]). This means that we need to include in the training dataset at least 32% of real functions so that the trained models are able to predict return types of code written by programmers not included in the training dataset.

Table 5.5 shows the accuracy, precision, recall and F_1-score of our models and the existing compilers, using the third evaluation method (Section 5.2.6). Gradient boosting is the classifier with the best results: 0.839 accuracy and 0.838 F_1-score. Sometimes, there are no significant differences between random forest and extremely randomized trees. Gradient boosting accuracy and F_1-score are, respectively, 43.9% and 101.9% higher than the best decompiler (IDA).

		Predicted class									
		`bool`	`char`	`short`	`int`	`pointer`	`struct`	`long long`	`float`	`double`	`void`
Actual class	`bool`	**483**	**61**	0	1	12	0	41	0	0	2
	`char`	**285**	**179**	60	16	13	0	45	0	0	2
	`short`	7	85	395	38	24	0	44	0	1	6
	`int`	4	38	67	**184**	**111**	**132**	59	0	0	5
	`pointer`	4	16	3	**48**	**365**	**106**	55	0	0	3
	`struct`	0	0	0	**10**	**6**	**584**	0	0	0	0
	`long long`	0	4	0	5	28	3	554	0	0	6
	`float`	0	1	1	0	16	0	40	358	184	0
	`double`	0	0	0	0	12	0	52	170	365	1
	`void`	3	2	0	2	3	0	41	0	0	549

Table 5.6: Confusion matrix for the decision tree classification of high-level types, with a balanced dataset comprising 6,000 functions.

5.3.2 Classifying with high-level types

In the previous subsection, we measured the performance of our predictive models, considering types by their size and representation. However, the objective of a decompiler is to infer the high-level C types, even if they share the exact size and representation. Following the same methodology, we now conduct a new experiment to reconstruct C types from binary code.

In this case, we consider the C built-in types `bool`, `char`, `short`, `int`, `long long`, `float`, `double` and `void`. For the particular case of Microsoft `cl` and 32-bit architecture, the `long` type is exactly the same as `int`, since the semantic analyzer allows the very same operations and its target size and representation are the same; the same occurs for `double` and `long double`. For this reason, `long` and `long double` types are considered the same as, respectively, `int` and `double`. The `signed` and `unsigned` type specifiers are not considered, as there is no difference between their binary representations.

We also consider `pointer` and `struct` type constructors. Arrays are not classified because C functions cannot return arrays (they actually return pointers) [92]. As discussed in Section 5.3.1, the `union` type constructor is actually represented as one single variable with the size of the biggest field, so `cl` generates no different code when the type of the biggest field is used instead of `union`. The same happens with `enum` and `int`.

Our classifiers detect the `struct` and `pointer` type constructors, but not the `struct` fields or the pointed type. Once we know the return type is `pointer` or `struct`, the subtypes used to build the composite types could be obtained with existing deterministic approaches [19, 23].

We first analyze how well the exiting pattern generalizations (Table 4.1) classify high-level return types. To that end, we conduct the following experiment. First, we define the target variable as the high-level C types described above. Then, we build a decision tree classifier and evaluate it with a balanced test dataset comprising 6,000 functions. The confusion matrix obtained is shown in Table 5.6. If we analyze the two types with 1-byte size (`bool` and `char`), we can see that 28.8% of the instances are misclassified between `bool` and `char`. A similar misclassification issue occurs for the 4-byte-size types `int`, `struct` and

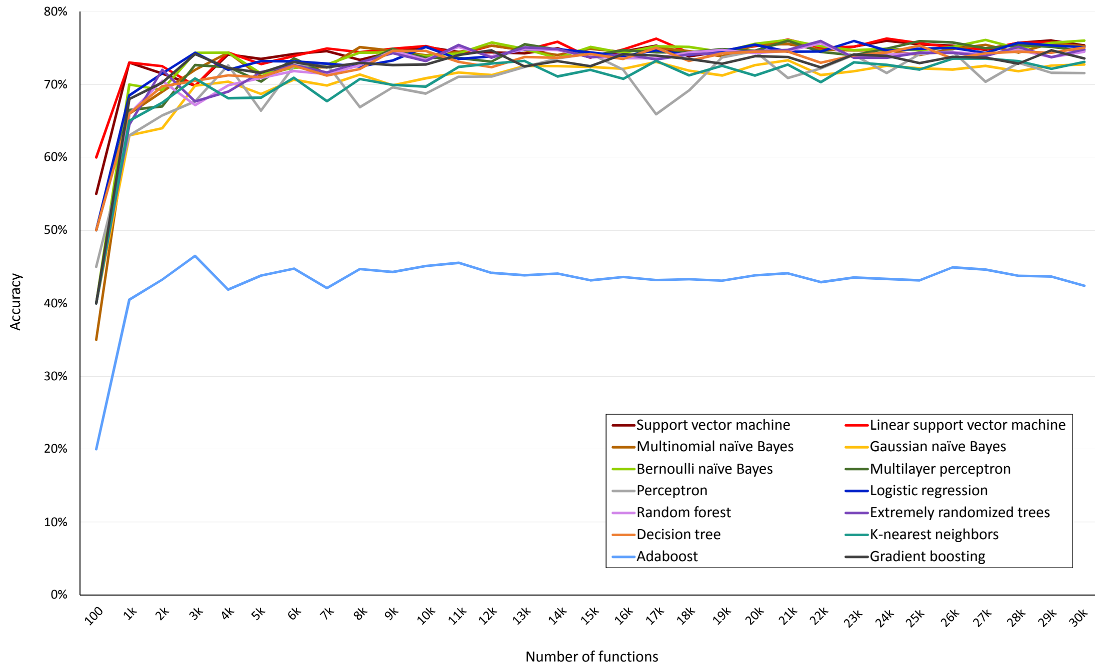


Figure 5.9: Classifiers accuracy for increasing number of functions (classifiers of high-level types).

Classifier	Applied method (wrapped algorithm, accuracy threshold)	Selected features
AdaBoost	SelectFromModel(Random forest, Mean)	147
Bernoulli naïve Bayes	SelectFromModel(Extremely randomized trees, Median)	456
Decision tree	SelectFromModel(Random forest, Median)	455
Extr. randomized trees	SelectFromModel(Random forest, Median)	455
Gaussian naïve Bayes	SelectFromModel(Extremely randomized trees, Median)	456
Gradient boosting	SelectFromModel(Extremely randomized trees, Median)	456
K-nearest neighbors	SelectFromModel(Random forest, Median)	455
Lin. sup. vector machine	SelectFromModel(Random forest, Median)	455
Logistic regression	SelectFromModel(Random forest, Median)	455
Multilayer perceptron	SelectFromModel(Extremely randomized trees, Median)	456
Multinomial naïve Bayes	SelectFromModel(Extremely randomized trees, Median)	456
Perceptron	SelectFromModel(Random forest, Median)	455
Random forest	SelectFromModel(Random forest, Median)	455
Support vector machine	SelectFromModel(Random forest, Median)	455

Table 5.7: Best feature selection method used for each classifier (classifiers of high-level types).

`pointer`, mistaking 22.9% of the instances among these three types.

This experiment shows us how there is still room for improving the classification of high-level types with similar size and representation. Aware of that, we apply the generalization method described in Section 4.3 to include new generalization patterns for differentiating among high-level types with the same size and representation. We search for the misclassified functions in Table 5.6 and analyze the sequences of assembly code related to the same type. Once detected, we generalize and include them in our pattern extractor (Figure 5.1) to improve the classifiers. Therefore, we use machine learning to detect some of the limitations of the existing classifiers, analyze potential binary patterns, define new features of the dataset, and create better models to classify the return type of decompiled functions (Section 4.3). The new generalizations defined are detailed in Appendix B.

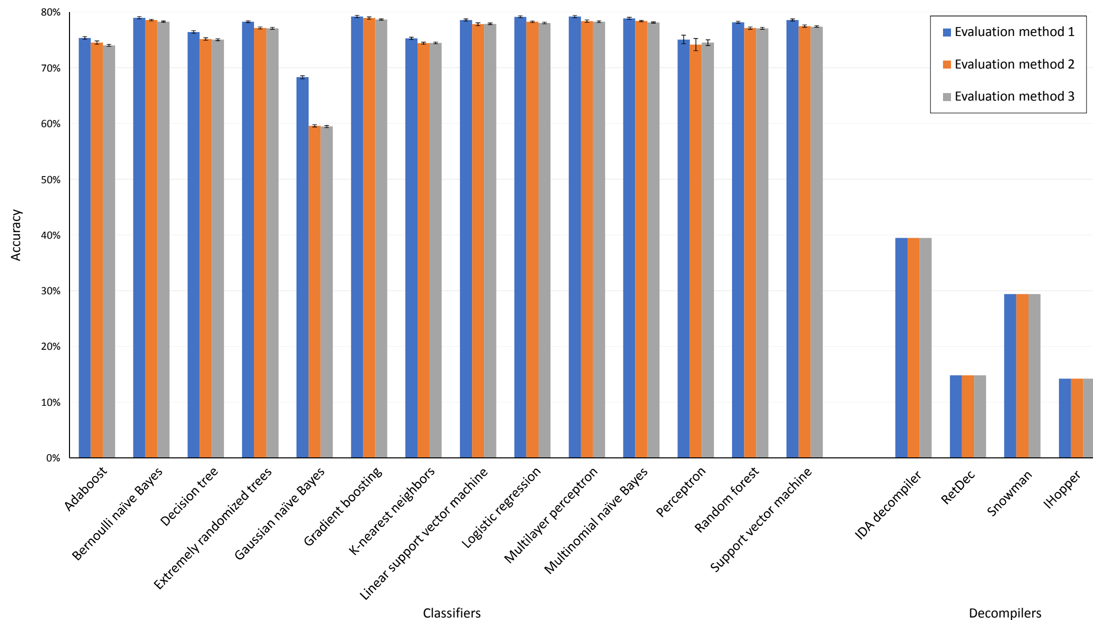


Figure 5.10: Accuracies of our classifiers and the existing decompilers, using the three different evaluation methods described in Section 5.2.6 (classifiers of high-level types).

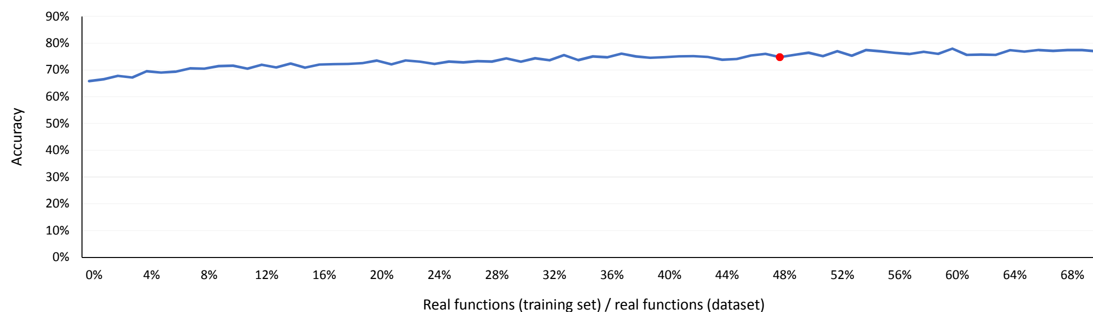


Figure 5.11: Accuracy of a decision tree for different percentage of real functions included in the training dataset (classifiers of high-level types). The red dot indicates the value where the CoV of the last 10 accuracies is lower than 1%.

A new dataset is created with all the new generalization features in Appendix B to classify the ten different high-level types mentioned. We compute the optimal size of the dataset with the algorithm described in Section 5.2.2. The resulting dataset has 18,000 synthetic functions (Figure 5.9) plus the 2,339 functions implemented by real programmers. Following the methodology described in Section 5.2, we run different feature selection algorithms, obtaining the features in Table 5.7. The existing 1,036 features were reduced, on average, to 433. We then tune the hyperparameters of the models, achieving similar values to the previous experiment (their values can be consulted in Appendix D).

5.3.2.1 Results

Figure 5.10 compares the accuracies of the new models and the selected decompilers (detailed data is depicted in Table 5.8). All the models outperform the existing decompilers. As in the previous case, the first evaluation method has significant differences with the two last ones (which obtain similar results). For

		Accuracy	Precision	Recall	F_1-score
Classifiers	AdaBoost	0.740 ± 0.49%	0.764 ± 0.92%	0.741 ± 0.42%	0.741 ± 0.65%
	Bernoulli naïve Bayes	0.783 ± 0.29%	**0.822** ± 0.33%	**0.785** ± 0.31%	**0.792** ± 0.31%
	Decision tree	0.750 ± 0.47%	0.772 ± 0.85%	0.753 ± 0.47%	0.751 ± 0.59%
	Extr. randomized trees	0.771 ± 0.42%	0.793 ± 0.82%	0.774 ± 0.45%	0.772 ± 0.56%
	Gaussian naïve Bayes	0.595 ± 0.67%	0.671 ± 0.88%	0.692 ± 0.35%	0.619 ± 0.51%
	Gradient boosting	**0.786** ± 0.31%	**0.820** ± 0.29%	0.783 ± 0.31%	**0.791** ± 0.30%
	K-nearest neighbors	0.745 ± 0.40%	0.759 ± 0.78%	0.749 ± 0.41%	0.743 ± 0.57%
	Lin. sup. vector machine	0.779 ± 0.42%	0.803 ± 0.81%	0.782 ± 0.34%	0.780 ± 0.59%
	Logistic regression	0.780 ± 0.36%	0.802 ± 0.58%	**0.785** ± 0.36%	0.785 ± 0.37%
	Multilayer perceptron	0.783 ± 0.36%	0.815 ± 0.65%	**0.784** ± 0.37%	0.788 ± 0.43%
	Multinomial naïve Bayes	0.781 ± 0.28%	0.818 ± 0.29%	**0.785** ± 0.31%	0.789 ± 0.30%
	Perceptron	0.745 ± 1.35%	0.789 ± 1.20%	0.747 ± 1.18%	0.749 ± 1.23%
	Random forest	0.771 ± 0.43%	0.796 ± 0.75%	0.773 ± 0.46%	0.773 ± 0.52%
	Support vector machine	0.774 ± 0.35%	0.816 ± 0.34%	0.779 ± 0.33%	0.784 ± 0.32%
Decomp.	IDA decompiler	0.40	0.33	0.34	0.30
	RetDec	0.15	0.06	0.10	0.06
	Snowman	0.29	0.22	0.26	0.21
	Hopper	0.14	0.08	0.09	0.03

Table 5.8: Performance of the classifiers and existing decompilers using the third evaluation method (classifiers of types with different size and representation). 95% confidence intervals are expressed as percentages. Bold font represents the best values. If one column has multiple cells in bold, it means that values are not significantly different.

		Predicted class									
		`bool`	`char`	`short`	`int`	`pointer`	`struct`	`long long`	`float`	`double`	`void`
Actual class	`bool`	**491**	**49**	0	8	10	0	39	0	0	3
	`char`	**231**	**212**	64	20	23	0	45	0	0	5
	`short`	6	76	379	45	36	0	49	0	0	9
	`int`	6	44	72	**279**	**111**	**21**	59	0	0	8
	`pointer`	12	8	0	**35**	**419**	**72**	47	0	0	7
	`struct`	0	0	0	**1**	**12**	**587**	0	0	0	0
	`long long`	5	8	0	6	28	4	543	0	1	5
	`float`	0	0	0	0	11	0	48	355	184	2
	`double`	0	0	0	0	12	0	44	177	365	2
	`void`	4	1	1	1	4	0	48	0	0	541

Table 5.9: Confusion matrix of the model in Table 5.6, including the generalizations of Appendix B.

the second method, we found that at least 48% of the real functions must be included in the training dataset to obtain accuracy convergence (Figure 5.11). With this percentage of real functions, our models are able to predict functions of programmers whose code is not included in the training set (*i.e.*, there are no significant differences between the second and third evaluation method).

Table 5.8 shows how gradient boosting obtains the best performance: 0.786 accuracy and 0.791 F_1-score for the third evaluation method. Comparing these values with the existing decompilers, the performance of gradient boosting is from 96.5% (accuracy) to 163.6% (F_1-score) higher than the decompiler with the highest performance (IDA). Therefore, the gradient boosting vs IDA benefit is increased by 90.1% when predicting high-level C types.

Table 5.9 shows the new values of the confusion matrix presented in Table 5.6, running the same experiment with the new generalizations. The performance gains obtained when classifying types with the same size and representation are summarized in Table 5.10. We obtain a 10.5% average F_1-score gain for 1-byte-

		Accuracy gain	Precision gain	Recall gain	F_1-score gain
INT_1	bool	0.84%	5.85%	1.66%	3.98%
	char	1.01%	14.86%	18.44%	17.01%
INT_4	int	1.81%	16.70%	51.63%	37.76%
	pointer	0.58%	1.69%	14.79%	7.90%
	struct	2.56%	21.23%	0.51%	11.55%

Table 5.10: Performance gains obtained for high-level type classification, when the generalizations in Appendix B are included in the dataset.

size types and 19.1% for types of 4 bytes, due to the additional generalizations detailed in Appendix B.

Chapter 6

Binary patterns found

We have implemented a platform for the automatic extraction of binary patterns (Chapter 3). We have also defined a method to generalize and use those patterns for decompilation (Chapter 4). Then, we have applied them for the particular purpose of predicting the high-level C type returned by functions (Chapter 5). Furthermore, the datasets created can be used to discover and document binary patterns to be included in existing decompilers. In this chapter, thus, we mine the dataset to document binary patterns associated with high-level return types. That documentation can be helpful to improve the implementation of current decompilers.

6.1 Association rules

We discover binary patterns with association rules that correlate RET and POST CALL patterns with return types of function. Since the dataset has a high number of features, we first select the most important features with the five feature-selection algorithms described in Section 5.2.4. We choose the intersection of the feature sets selected by those algorithms. Then, we run the Apriori algorithm for association rule mining [93], saving the rules whose consequent is a return type.

In this chapter, we only analyze the rules with 100% confidence. The confidence of an association rule measures how frequently the rule consequent holds when the antecedent is true. In this way, the association rules retrieved represent a mechanism to document those RET and POST CALL binary patterns that are unambiguously associated with a high-level return type.

	Antecedents	Consequent	Support
1	(RET) `mov al`, *literal* *callee_epilogue* (POST CALL) *caller_epilogue* `movzx edx, al` where: *literal* ∈ {`0`, `1`}	`bool`	0.00059
2	(RET) `mov al`, *literal* *callee_epilogue* where: *literal* ∈ {`0`, `1`}	`char`	0.00536
3	(RET) *binary_op* `eax` arg_1 *callee_epilogue* (POST CALL) *caller_epilogue* *mov* arg_2, `al` where: *binary_op* ∈ {`imul`, `sub`, `add`, `sar`, `sal`, `shr`, `shl`, `xor`, `or`, `and`} arg_1 ∈ {*reg*, `[`*reg*`]`, **address*, *literal*} *mov* ∈ {`mov`, `movzx`} arg_2 ∈ {*reg*, `[`*reg*`]`, **address*}	`char`	0.00207
4	(RET) `idiv` arg_1 *callee_epilogue* (POST CALL) *caller_epilogue* *mov* arg_2, `al` where: arg_1 ∈ {*reg*, `[`*reg*`]`, **address*} *mov* ∈ {`mov`, `movzx`} arg_2 ∈ {*reg*, `[`*reg*`]`, **address*}	`char`	0.00030
5	(RET) *unary_op* `eax` *callee_epilogue* (POST CALL) *caller_epilogue* *mov arg*, `al` where: *unary_op* ∈ {`not`, `neg`} *mov* ∈ {`mov`, `movzx`} *arg* ∈ {*reg*, `[`*reg*`]`, **address*}	`char`	0.00049
6	(POST CALL) *caller_epilogue* `cwde`	`short`	0.01190
7	(POST CALL) *caller_epilogue* *mov arg*, `ax` where: *mov* ∈ {`mov`, `movzx`, `movsx`} *arg* ∈ {`eax`, `ecx`, `edx`, `cx`, `si`, `[`*reg*`]`, **address*}	`short`	0.02455
8	(RET) *mov* `ax`, *arg* *callee_epilogue* where: *mov* ∈ {`mov`, `movzx`, `movsx`} *arg* ∈ {`dx`, `al`, `cx`, `cl`, `[`*reg*`]`, **address*}	`short`	0.00871

(continues)

	Antecedents	Consequent	Support

9	(RET)
	cond_jmp $offset_1$
	mov [reg_1], $literal_1$
	jmp $offset_2$
	mov [reg_1], $literal_2$
	mov eax, [reg_1]
	callee_epilogue
	
	where: *cond_jmp* ∈ {jo, jno, js, jns, je, jz, jne, jnz, jb, jnae, jc,
	↪ jnb, jae, jnc, jbe, jna, ja, jnbe, jl, jnge, jge, jnl, jle,
	↪ jng, jg, jnle, jp, jpe, jnp, jpo, jcxz, jecxz}
	$literal_1$, $literal_2$ ∈ {0, 1}
	$literal_1 \neq literal_2$
	$offset_1 \neq offset_2$
		int	0.01594

10	(RET) *div* ecx mov eax, edx *callee_epilogue* where: *div* ∈ {div, idiv}	int	0.00103

11	(RET)
	mov eax, *literal*
	callee_epilogue
	
	(POST CALL)
	caller_epilogue
	mov *arg*, eax
	
	where: *arg* ∈ {[*reg*], **address*}
		int	0.00221

12	(RET) *binary_op* eax, *address* *callee_epilogue* where: *binary_op* ∈ {mov, movzx, movsx, add, sub}	pointer	0.00949
13	(RET) lea eax, [*reg*] *callee_epilogue*	pointer	0.01304
14	(RET) mov eax, [ebp+8] *callee_epilogue*	struct	0.06321
15	(RET) cdq *callee_epilogue*	long long	0.04442
16	(RET) mov edx, *arg* *callee_epilogue* where: *arg* ∈ {*reg*, [*reg*], **address*, *literal*}	long long	0.01063
17	(RET) fstp [*reg*] fld [*reg*] *callee_epilogue* where: opcode(fstp)[0] = 0xD9 opcode(fld)[0] = 0xD9	float	0.03817
18	(POST CALL) *caller_epilogue* fstp *arg* where: opcode(fstp)[0] = 0xD9 *arg* ∈ {[*reg*], **address*}	float	0.01545

(continues)

	Antecedents	Consequent	Support
19	(RET) `fld` *arg* *callee_epilogue* where: opcode(`fld`)[0] = 0xDD *arg* ∈ {`[`*reg*`]`, **address*}	`double`	0.06783
20	(RET) `mov` arg_1 arg_2 *callee_epilogue* where: arg_1 ∈ {`[`*reg*`]`, **address*} arg_2 ∈ {*reg*, `[`*reg*`]`, **address*, *literal*}	`void`	0.00817

Table 6.1: Example association rules obtained from the dataset. *reg* variables represent registers, *literal* integer literals, *address* absolute addresses, **address* absolute addresses dereferences and *offset* relative addresses.

Table 6.1 shows some of the rules obtained with at most two antecedents. The rest of them are detailed in Appendix A. The support of each rule is the relative frequency of instances covered by a rule. Rules with very low support are not included in Table 6.1.

The `cdecl` calling convention [94] returns 32-bit values through `eax`; `ax` is used for 16-bit values, and `al` for 8-bits. 64-bit integers are returned via `edx` and `eax` registers. The 32- and 64-bit real values are returned through `st0`. The main problem is to determine whether such registers are actually returning their values to the caller, or they just hold temporary values of previous computations. Another important problem, as mentioned, is to classify the high-level types with equal size and representation.

Rules 1-5 return the value with `al`, so they classify `bool` and `char` types. The `ax` register in rules 6-8 is used to return `short`. Rules 9-14 analyze `eax` for 4-byte-size types. Rules 15 and 16 check `edx` to infer `long long`, and rules 17-19 use `st0` to return `float` and `double`. The last rule checks that the value copied before returning from the function call is not moved to a register (but to a memory address), classifying the function type as `void`.

Some functions return literal values, such as `true` or `32`. The value of those literals is used by some patterns to infer the return type. For example, rule 2 classifies as `char` the 1-byte type returned when the returned literal is neither `0` nor `1` (low-level representation of `false` and `true`). The opposite is not true; when `0` or `1` is returned, it could be a character (`'\0'` character is widely used in C). For this reason, rule 1 adds a POST CALL check after the invocation. If `0` or `1` is returned and it is moved to `edx` with zero extension using `movzx` (*i.e.*, high bits are set to zero, without sign), the type is `bool`; for chars, `movsx` is used instead (copy with sign). Likewise, rule 12 uses address literals to classify pointers.

Our system also detects operations that can only be applied to certain types. For example, division (`div` and `idiv`) can be applied to neither `pointer` nor `struct`. Therefore, rule 10 infers a 4-byte type to `int`, when division operations

Textual representation	Opcodes	
	Float	Double
fstp reg_{fp}	D9 ?? ??	DD ?? ??
fld reg_{fp}	D9 ?? ??	DD ?? ??

Table 6.2: Binary encodings of fld and fstp instructions.

are applied to it. Rule 13 classifies as pointer any 4-byte type where a lea instruction is used, since lea loads a memory address into the target register (eax).

Another classification mechanism used by our models is based on the binary representation of assembly instructions. For example, the fstp and fld assembly instructions for real numbers share the start of the binary opcode[1] (Table 6.2). When they operate with 32-bit floating-point numbers, the opcode starts with 0xD9. However, their opcode starts with 0xDD when applied to 64-bit operands. This difference is used by rules 17-19 to tell the difference between float and double.

The classifiers generated with our dataset also detect binary patterns of the code generation templates implemented by compilers [74]. For example, rule 9 detects the code generation template used by cl to return the result of a comparison as an int. Of course, these kinds of templates are compiler dependent, so the compiler used should be discovered before using them [36] as indicated in Section 4.2.

Rule 14 is another rule for a particular code generation template. As described in Section 5.3.1, cl performs a code transformation to return struct types (Figure 5.4). The struct is passed as an argument, and its memory address is actually returned as a pointer. This code transformation generates a particular sequence of assembly instructions that our models use to identify structs among types of 4-bytes size.

Although the assembly instructions used to return a value (RET patterns) are very important to infer return types, the binary code used after the invocation (POST CALL patterns) is also valuable. For example, rules 6 and 7 identify as short type the usage of ax just after an invocation. Another example is rule 18, which stores the returned floating-point value from the mathematical coprocessor stack.

Finally, our system is also able to combine RET and POST CALL patterns to infer return types. Since these kinds of rules are more specific, they commonly have low support and high confidence. For example, the best rule found to classify bool with one RET pattern provides 76% confidence; whereas rule 1 provides 100% confidence by adding a POST CALL pattern with lower support. These types of rules commonly classify types among others with similar size and representation, such as rules 1, 3 and 4 (1 byte), and rule 11 (4 bytes).

[1] Opcode stands for operation code. It is the portion of the numeric representation of an assembly instruction that specifies the operation to be performed.

All the association rules with at most three antecedents, minimum confidence of 95%, and support greater than 0.02% are detailed in Appendix A.

Chapter 7

Controlled stochastic generation of standard C source code

Big code works use the source code of millions of programs available in open source-code repositories to build different types of tools [95]. However, obtaining portable C code is not an easy task, since most applications use particular libraries or operating system. In addition, there also exist many different variants of the C language, which include language extensions and modifications. Therefore, different ANSI/ISO standardizations of C are defined to facilitate the development of portable software [14]. Even so, it is still difficult to find applications written in 100% standard C source code. Most of them have particular dependencies of non-portable code. This is an issue when building predictive models from source code since a large number of programs is usually required [96].

For these reasons, we implement Cnerator, a Python application for the controlled stochastic generation of standard C source code. Cnerator provides the generation of large amounts of standard ANSI/ISO C source code [14], ready to be compiled by any standard language implementation. Cnerator is highly customizable to generate all the syntactic constructs of the C language, necessary to build accurate predictive models with machine learning algorithms. The stochastic generation of source code programs has also been used to detect bugs in existing compilers [97]. Another potential use of Cnerator is testing whether a compiler implements the ANSI/ISO standard specification correctly.

7.1 Software framework

In this section, we first describe the main functionalities of Cnerator. Then, we present its architecture and a brief description of each module.

7.1.1 Software functionalities

These are the main functionalities provided by Cnerator:

1. ANSI/ISO standard C. All the source code generated by Cnerator follows the ISO/IEC 9899:2018 (C17) standard specification [14].

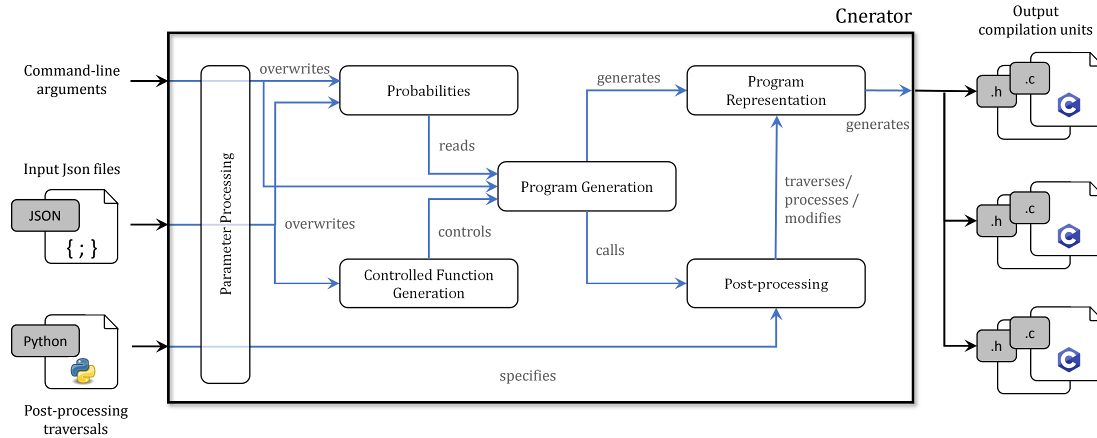


Figure 7.1: Architecture of Cnerator.

2. Probabilistic randomness. C language constructs are randomly generated, following different probability distributions specified by the user. For example, it is possible to describe the probability of each kind of statement and expression construct, the number of statements in a function, and the types of their arguments and return values. To this aim, the user can specify fixed probabilities of each element, or use different probability distributions, such as normal, uniform, and direct and inverse proportional.

3. Compilable code. The generated code strictly follows the syntax grammar, type system, and semantic rules of the C programming language. In this way, the generated code has been checked to be compilable by any standard compiler implementation.

4. Highly customizable. Many features of the programs to be generated are customizable. Some examples include the types of each language construct, array dimensions and sizes, struct fields, maximum depth of expression and statement trees, number of function parameters and statements, global and local variables, structures of control flow statements, and type promotions, among others –see the detailed documentation Appendix C.

5. Large amounts of code. Cnerator is designed to allow generating large amounts of C source code. A parameter indicates the number of independent compilation units to be created for the output application, so that each unit could be treated as an independently compilable module. This feature, together with the probabilistic randomness, make Cnerator an ideal tool to build predictive models, because the input programs used to train such models comprise abundant and varied code patterns.

7.1.2 Architecture

Figure 7.1 presents the architecture of Cnerator. When executing the tool, three types of optional arguments may be passed: *command-line arguments*, *JSON specification files* and *Python post-processing traversals*. If no parameter is passed, Cnerator creates a random output program, using the default probability values (Appendix C). The generated program consists of a group of *compilation units*

(a pair of `.h` and `.c` files) that can be compiled independently, even though they commonly depend on other compilation units.

As *command-line arguments*, the user may pass parameters such as the number of output compilation units, probability values of syntactic constructs, and the output directory and file names, among others (all the parameters are detailed in Appendix C). The *Parameter Processing* module takes all the parameters passed by the user and customizes the behavior of Cnerator accordingly.

Cnerator accepts two types of JSON configuration files as parameters (examples are presented in Section 7.2). The first one allows specifying the probability values and probability distributions of multiple C syntactic constructs. The *Probabilities* module stores the default probability distributions of all the syntax constructs and provides different helper functions to facilitate its specification. As shown in Figure 7.2 (explained in Section 7.2), JSON probability specification files permit the use of those helper functions to modify the default probability distributions.

The second type of JSON input allows the user to control the number and characteristics of all the functions to be generated. For example, we can enforce Cnerator to generate a program with as many functions as built-in types in the language, and make each function return an expression of each built-in type. The *Controlled Function Generation* module interprets the JSON file to drive the process of program generation. To this aim, it asks the main *Program Generation* module to generate random functions, and discards those not fulfilling the requirements specified in the JSON file. If no function generation file is provided, *Program Generation* just produces a random program following the existing probability distributions.

The third type of argument is an ordered collection of *Python post-processing specification files*. When the user wants the output program to fulfill some requirements not guaranteed by the stochastic generation process, these post-processing files can be used to modify the generated code in order to meet such requirements. By following an introspective implementation of the Visitor design pattern [98], the user can specify the traversal of the program representation produced by Cnerator (an example is presented in Section 7.2). We use the `single-dispatch` Python package [99] to traverse program representations.

The *Program Representation* module is mainly an in-memory representation of Abstract Syntax Trees (AST) [100]. Cnerator produces ASTs modeling the generated program before generating the output code. The AST data structure implements the Interpreter design pattern [101] to convert a program representation into a set of *output compilation units*.

7.2 Illustrative example

In this section, we show how Cnerator was used to generate the synthetic code used in Chapter 5 to build the decompilation models. Figure 7.2 shows an excerpt of two of the JSON files used to customize Cnerator. The one on the left over-

```
{
   "function_basic_stmt_prob": {
      "assignment": 0.3,
      "invocation": 0.4,
      "augmented_assignment": 0.15,
      "incdec": 0.1,
      "expression_stmt": 0.05
   },
   "array_literal_initialization_prob": {
      "True": 0.1, "False": 0.9
   },
   "primitive_types_prob": {
      "__prob_distribution__": "equal_prob",
      "__values__": [
         "ast.Bool",
         "ast.SignedChar",
         "ast.UnsignedChar",
         "ast.SignedShortInt",
         …
      ]
   },
   "param_number_prob": {
      "__prob_distribution__":
       ↪                 "proportional_prob",
      "__values__": {
         "0": 1, "1": 2, "2": 3,
         "3": 3, "4": 2, "5": 1
      }
   },
   "number_stmts_main_prob": {
      "__prob_distribution__": "normal_prob",
      "__mean__": 10,
      "__stdev__": 3
   },
   …
}
```

```
{
  "function_returning_void": {
    "total": 1000,
    "condition": "lambda f:
     ↪       isinstance(f.return_type,
     ↪                  ast.Void)"
  },
  "function_returning_bool": {
    "total": 1000,
    "condition": "lambda f:
     ↪       isinstance(f.return_type,
     ↪                  ast.Bool)"
  },

  …

  "function_with_if_else": {
    "total": 1,
    "condition": "lambda f:
     ↪ any(stmt for stmt in f.children
     ↪    if isinstance(stmt, cnerator.ast.If)
     ↪        and any(stmt.else_statements))"
  }
}
```

Figure 7.2: Two example JSON files used to customize program generation with Cnerator. The left-hand side shows a sample probability specification file, and the right-hand side specifies an example of controlled function generation.

```
01: from singledispatch import singledispatch
02: from cnerator import ast
03:
04: def _instrument_statements(statements: List[ast.ASTNode]) -> List[ast.ASTNode]:
05:     """Includes a unique label before any return statement"""
06:     instrumented_stmts = []
07:     for stmt in statements:                          # iterates through the statements
08:         if isinstance(stmt, ast.Return):             # if the statement is return…
09:             label_ast = ast.Label(generate_label())  # creates a new AST node for the label
10:             instrumented_stmts.append(label_ast)     # and places the label before the return
11:         visit(stmt)                                  # traverses the statement
12:         instrumented_stmts.append(stmt)              # appends return after the label
13:     return instrumented_stmts
14:
15: @visit.register(ast.Function)
16: def _(function: ast.Function):
17:     """Traverses a function definition to add a unique label before each return statement"""
18:     function.stmts = _instrument_statements(function.stmts)
19:
20: @visit.register(ast.Do)
21: @visit.register(ast.While)
22: @visit.register(ast.For)
23: @visit.register(ast.Block)
24: def _(node):
25:     """Traverses a control flow statement to add a unique label before each return statement"""
26:     node.statements = _instrument_statements(node.statements)
27:
28: _return_label_counter = 0
29:
30: def generate_label() -> str:
31:     """Generates a new unique label string"""
32:     global _return_label_counter
33:     _return_label_counter += 1
34:     return f"__RETURN{_return_label_counter}__"

    …
```

Figure 7.3: Python code excerpt of an AST post-processing example.

writes some default probabilities. The first entry (`function_basic_stmt_prob`) defines the probability of building basic statements (*i.e.*, statements not containing other statements, unlike `for` and `switch`), and the second one states that 10% array definitions should initialize their values. These two examples specify fixed probabilities that must sum zero. The three remaining entries use uniform/equal, proportional and normal distributions to specify, respectively, the usage of primitive types, and the number of function parameters and statements in the main function.

The right-hand JSON file in Figure 7.2 shows the controlled function-generation method used to build the decompilation models. The two first entries are examples of how we made Cnerator generate 1000 functions returning each type provided by standard C[1]. The condition in the `lambda` expression checks that the returned type is the expected one. The last entry shows a different example, not used in the decompiler scenario, where the user demands Cnerator to generate a function containing an `if` statement with an `else` clause.

Figure 7.3 shows an example Python post-process specification file. The code traverses the representation (AST) of the generated program and adds a unique label before each `return` statement. The purpose of this instrumentation is to identify in the compiled code the binary patterns used for each high-level `return` statement (Section 5.1.1). Those binary code patterns are later labeled with

[1]Only `void` and `bool` are shown for the sake of brevity.

the high-level return type to build predictive models with supervised machine learning (Chapters 3 and 5).

The `_instrument_statements` function takes a list of statements (represented as AST nodes) and adds a unique label –prefixed with `__RETURN` (line 09)– before each return statement. That function is later used in the traversal of function definitions (line 18), and `do`, `while`, `for` and block statements (line 26) –`if` and `switch` control flow statements follow the same template. The code in Figure 7.3 is an instance of an introspective implementation of the Visitor design pattern [98]. The `visit` annotations indicate the AST node to be traversed, and default tree traversal is performed with reflection [102].

Chapter 8

Conclusions

We have seen how machine learning can be used to decompile high-level source code from binaries, providing a significant improvement of the state-of-the-art decompilers. Supervised machine learning is useful not only to build predictive models by processing large amounts of binaries, but also as a feature engineering tool to generalize binary patterns.

An efficient platform for the automatic extraction of patterns in binary code has been designed and implemented. This platform allows the user to declaratively obtain different kinds of patterns from massive binary codebases. The high-level language constructs are associated with the binary code, compiled, extracted, and labeled to construct the dataset used for building the predictive models. To speedup binary pattern extraction, the implementation is highly parallelized, following a pipeline approach with both data and task parallelization. We obtain a speedup of 3.5 factors in a 4-core computer, when the maximum theoretical benefit is of 4 factors.

We propose a method to create predictive models for decompilation. First, we identify all the variables that influence the decompilation models: compiler used, binary file format, operating system, word size, compiler options, and target microprocessor. Our method describes how to face the high dependency of these variables on the decompilation process. Second, we define a pattern generalization process to improve the performance of machine-learning models and to deal with the huge variability of binary code. Third, we describe a method that, using machine learning, assists in the creation of decompilation models for different language constructs.

It is difficult to obtain a huge database of standard C source code to be compiled by any compiler, because most projects have strong dependencies on a particular operating system or language extension. For this reason, we designed and implemented Cnerator, an application for the controlled stochastic generation of standard C source code. It provides the generation of creating large amounts of standard ANSI/ISO C source code, ready to be compiled by any standard language implementation. Cnerator is highly customizable to generate all the syntactic constructs of the C language, necessary to build accurate predictive models with machine learning algorithms.

We take the platform to extract binary patterns, the Cnerator tool to generate a collection of C source code programs, some “real” open-source projects, and apply the proposed method to the decompilation problem of inferring the high-level type returned by functions. The predictive models created with this process obtain a 79.1% F_1-score, while the best existing decompiler provides a 30% F_1-score. This value is measured with source code of programmers not included in the training dataset. The combination of “real” and synthetic code makes our models be able to predict type information of unseen programming styles.

Finally, we discuss and document the binary patterns found to classify return types. The classification rules combine binary patterns extracted from return expressions inside the function bodies and the binary code used just after the invocations to those functions. They focus on how data are passed from the function to the caller, and how they are later used. The binary patterns found not only distinguish among different sizes and representations of data, but also among types with the same binary size and representation. These patterns could be included in existing decompilers to improve their accuracy for inferring return types.

Chapter 9

Future work

The work presented in this dissertation opens new future lines of research that we plan to work on. What follows is a brief description of such works.

9.1 Automatic detection of influencing variables

As we have seen in Section 4.2, the predictive models aimed at improving decompilation show a strong dependency on different variables, such as compiler options, binary file formats, and target operating system and microprocessor. Accordingly, we propose training different models for different combinations of these influencing variables. Therefore, the problem would be how to detect the particular values of such variables from binary files, when they include no debugging information.

As mentioned in Chapter 4, some of these influencing variables, such as binary file format, target microprocessor, and word size can be inferred deterministically [76]. For the rest of the variables, there exist some attempts to create models to detect the compiler used [36], the particular compilation options [37], and the target operating system [77]. We plan to analyze these techniques, improve them in case it is necessary, and integrate them into our system to allow the coexistence of different models that allow decompilation for different combinations of those influencing variables.

9.2 Decompilation of other native languages

Chapter 4 describes a method to decompile any language construct, and Chapter 5 applies it to reconstruct the high-level type returned by functions. We plan to use the same method for other language constructs. First, we can start by inferring types in other contexts. A good candidate could be function parameters. The idea is to extract patterns before function invocation and inside the body of the invoked function. A def/use analysis would relate each binary pattern with its corresponding parameter to create the dataset. A similar approach could be used to recover the types of local and global variables.

Another case scenario where the method in Chapter 4 can be used is the decompilation of control flow statements. The models should be able to tell the difference between iterative (`while`, `for` and `do`) and conditional control flow statements (`if` and `switch`).

Finally, we could use our approach to improve the readability of the high-level code generated by the decompiler. For example, the name of variables is completely lost when the compiler generates binary code. However, depending on the use of each variable, it is useful to choose an informative name for each variable [6]. Those names could be learned from massive codebases, using the identifiers employed by most programmers. This would not recover the original identifier used by the programmer, but it will improve the readability of the generated code.

9.3 Inference of other language constructs

As most decompilers, we have chosen C as the output programming language for decompilation. Its "medium-level" of abstraction, together with its widespread usage, makes it to be the *lingua franca* for most decompilers (Section 4.4). However, it could be very interesting to apply the method detailed in Chapter 4 to decompile other native languages.

One language to be considered is C++, as it is a superset of C. To extend the work presented in Chapter 5, dynamic binding invocations of virtual methods must be detected and decompiled [50, 51, 103]. Such invocations are translated into binary indirect calls (*e.g.*, `call dword ptr [eax]`) that use the content of a register to locate the invoked function at runtime. This information must be processed to create the dataset.

Pascal, Go and Rust do not stick to the C ABI (Application Binary Interface), so their compilers do not use the same code generation patterns as C compilers. We would like to see to what extend the method proposed in Chapter 4 is could be applied for this kind of languages. New binary patterns and generalizations would need to be searched.

9.4 Graph neural networks

Recurrent neural networks (RNN) are a type of artificial neural network that processes sequential or time-series data [15]. This kind of topology has been successfully used to predict some high-level constructs from a sequence of binary instructions, including the number and types of function parameters [10] and some code snippets [12]. RNNs process a sequence of inputs to compute a hidden state for each particular sequence, which is later used to perform some predictions. When combined with language models, they have successfully used to perform statistical machine translation with encoder-decoder topologies [104].

The classical definition of RNN describes a one-dimensional sequence of inputs. However, there are problems where those sequences require to have mul-

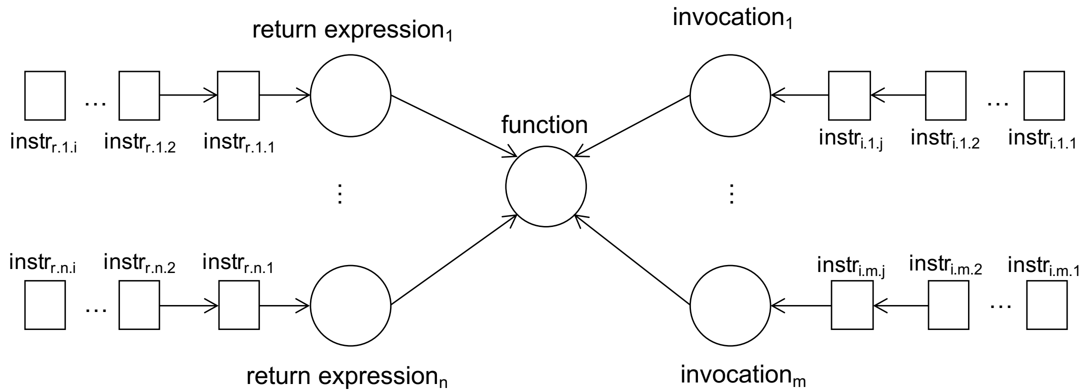


Figure 9.1: Graph neural network for inferring the high-level type returned by functions.

tiple dimensions. In such cases, graph neural networks (GNN) provide those multi-dimensional inputs by accepting graph structures as inputs [105]. GNNs support node-, edge-, and graph-level classification [106]. One hyperparameter is the depth of neighbors considered to compute the hidden state of a given node.

Figure 9.1 shows how we plan to classify function return types with GNNs. The node to classify is the function itself (its return type), which is connected with all the RET (`return` expressions) and POST CALL (invocations) binary patterns described in Chapter 3 of this dissertation. These patterns represent a graph-based structure connected to the function node. Moreover, the different sequences of binary instructions for each pattern (*inst* rectangles in Figure 9.1) represent a valuable one-dimensional sequence input to compute the hidden state of invocation and return nodes in the graph. With all this information, we think we can create deep learning predictive models with high accuracy, using massive inputs produced by Cnerator.

9.5 Addition of patterns to existing decompilers

In Chapter 6 and Appendix A we list some rules that associate RET and POST CALL binary patterns with the return type of functions. These rules distinguish not only between different sizes and representations of data but also among types with the same binary size and representation. All this new knowledge discovered in this dissertation could be included in the implementation of existing decompilers.

An IDA decompiler plugin could be implemented with the IDAPython API [65] or the IDA SDK [107]. That plugin would read a binary file and apply those rules to improve the information inferred by the decompiler. Hopper can be extended through its SDK and it also provides a Python interface. Finally, the Snowman [52] and RetDec [55] decompilers are open-source projects, so the pull request service in their GitHub repositories could be used to include new decompilation rules.

9.6 Adding additional knowledge with inductive logic programming

Inductive Logic Programming (ILP) is sometimes referred to as the intersection between inductive machine learning and logic programming [108]. Machine learning is used to infer hypotheses from observations, synthesizing new knowledge from experience. Such knowledge is represented as first-order logic, coded as a logic program.

One of the distinguishing features of ILP compared to traditional machine-learning algorithms is that ILP can use background knowledge for the learning process. In the particular problem of decompilation, this characteristic seems to be very promising because it allows improving our proposed method with alternative sources of information. First, decompilation experts can express their knowledge for decompiling a syntax construct with first-order logic. Second, we can provide to the ILP algorithm other sources of information extracted from the static analysis of the assembly code such as symbolic execution or any other deterministic approach [19, 21, 22, 23].

Unfortunately, the expressiveness increase of ILP is commonly associated with higher computational requirements, because the search space is significantly higher than traditional approaches (hypotheses are expressed with predicates instead of with propositions). Some performance experiments have indicated that ILP does not seem to scale well with big datasets [109].

Appendix A

Association rules found

This appendix shows a complete set of valid association rules that correlate RET and POST CALL patterns (Section 5.1.3.1) with the high-level return type of functions (Section 5.3.2). The rules analyzed in Chapter 6 are a condensed version of some of these rules.

To select the association rules, we filter the most important features with the 5 feature-selection algorithms described in Section 5.2.4. Later, we choose the intersection of all the feature sets. After that, we create a set of 26,239 association rules using the Apriori algorithm [93]. Finally, we select those rules with a minimum confidence of 95%, a minimum support of 0.0002 and 3 antecedents at most. The 6 possible formats are the following ones:

- {RET} ⇒ {*returned_type*}
- {POST_CALL} ⇒ {*returned_type*}
- {RET, POST_CALL} ⇒ {*returned_type*}
- {POST_CALL, POST_CALL} ⇒ {*returned_type*}
- {RET, POST_CALL, POST_CALL} ⇒ {*returned_type*}
- {POST_CALL, POST_CALL, POST_CALL} ⇒ {*returned_type*}

In the following tables, *reg* variables represent registers, *literal* integer literals, *address* absolute addresses, **address* absolute addresses dereferences and *offset* relative addresses.

A.1 Class `bool`

Antecedents	**Consequent**	**Confidence**	**Support**

(RET)
return_bool_cast `al, [`reg_{bp}`]`
callee_epilogue
(POST_CALL)
caller_epilogue
`movzx ecx, al`
(POST_CALL)
caller_epilogue
`movzx edx, al`
	`bool`	1	0.00113

(RET) *return_bool_literal* `al` *callee_epilogue* (POST_CALL) *caller_epilogue* `movzx edx, al`	`bool`	1	0.00059

(RET)
return_bool_cast `al, [`reg_{bp}`]`
callee_epilogue
(POST_CALL)
caller_epilogue
`mov` *address*`, al`
(POST_CALL)
caller_epilogue
`movzx eax, al`
	`bool`	1	0.00059

(POST_CALL)
caller_epilogue
`mov [`reg_{bp}`], al`
(POST_CALL)
caller_epilogue
`movzx eax, al`
(POST_CALL)
caller_epilogue
`movzx edx, al`
	`bool`	1	0.00049

(POST_CALL)
caller_epilogue
`mov` *address*`, al`
(POST_CALL)
caller_epilogue
`movzx eax, al`
(POST_CALL)
caller_epilogue
`movzx ecx, al`
	`bool`	1	0.00039

(POST_CALL)
caller_epilogue
`mov` *address*`, al`
(POST_CALL)
caller_epilogue
`movzx eax, al`
(POST_CALL)
caller_epilogue
`movzx edx, al`
	`bool`	1	0.00039

(RET)
`xor al, al`
callee_epilogue
(POST_CALL)
caller_epilogue
`mov [`reg_{bp}`], al`
	`bool`	1	0.00030

continues

Antecedents	Consequent	Confidence	Support

(POST_CALL)
caller_epilogue
`mov` *address*`, al`
(POST_CALL)
caller_epilogue
`movzx eax, al`
	`bool`	0.964	0.00133

(RET)
return_bool_cast `al, [`reg_{bp}`]`
callee_epilogue
(POST_CALL)
caller_epilogue
`movzx eax, al`
(POST_CALL)
caller_epilogue
`movzx edx, al`
	`bool`	0.95	0.00093

A.2 Class `char`

Antecedents	Consequent	Confidence	Support
(RET) *assign_char_literal* *callee_epilogue*	`char`	1	0.00536
(RET) *return_int_math_op* `al, ecx` *callee_epilogue*	`char`	1	0.00074

A.3 Class `short`

Antecedents	Consequent	Confidence	Support
(RET) *return_bool_cast* `ax, [`reg_{bp}`]` *callee_epilogue*	`short`	1	0.01864
(POST_CALL) *caller_epilogue* `cwde`	`short`	1	0.01190
(POST_CALL) *caller_epilogue* `movsx ecx, ax`	`short`	1	0.00644
(POST_CALL) *caller_epilogue* `movzx eax, ax`	`short`	1	0.00418
(POST_CALL) *caller_epilogue* `movzx ecx, ax`	`short`	1	0.00408
(POST_CALL) *caller_epilogue* `movzx edx, ax`	`short`	1	0.00384
(POST_CALL) *caller_epilogue* `mov [`reg_{bp}`], ax`	`short`	1	0.00330

continues

Antecedents	**Consequent**	**Confidence**	**Support**
(POST_CALL) *caller_epilogue* `mov` *address*`, ax`	`short`	1	0.00271
(RET) `movzx ax, [`reg_{bp}`]` *callee_epilogue*	`short`	1	0.00207
(RET) `mov ax, [`reg_{bp}`]` *callee_epilogue*	`short`	1	0.00182
(RET) `mov ax, [`reg_{ax}`]` *callee_epilogue*	`short`	1	0.00157
(RET) `mov ax, [`reg_{dx}`]` *callee_epilogue*	`short`	1	0.00138
(RET) *return_int_math_op* `ax, ecx` *callee_epilogue*	`short`	1	0.00123
(RET) `mov ax, [`reg_{cx}`]` *callee_epilogue*	`short`	1	0.00108
(RET) `mov ax,` *address* *callee_epilogue*	`short`	1	0.00079

A.4 Class `int`

Antecedents	**Consequent**	**Confidence**	**Support**
(RET) *return_bool_cast* `eax, [`reg_{bp}`]` *callee_epilogue*	`int`	1	0.01594
(RET) `mov eax,` *literal* *callee_epilogue* (POST_CALL) *caller_epilogue* `push eax`	`int`	1	0.00212
(RET) `mov eax,` *literal* *callee_epilogue* (POST_CALL) *caller_epilogue* `mov [`reg_{bp}`], eax`	`int`	1	0.00157
(RET) *return_int_math_op* `eax, ecx` *callee_epilogue*	`int`	1	0.00103
(RET) `mov eax,` *literal* *callee_epilogue* (POST_CALL) *caller_epilogue* `mov` *address*`, eax`	`int`	1	0.00064

continues

Antecedents	Consequent	Confidence	Support

(RET)
`xor eax, eax`
callee_epilogue
(POST_CALL)
caller_epilogue
`mov` *address*`, eax`
	`int`	1	0.00049

(RET)
`movsx eax, [`reg_{bp}`]`
callee_epilogue
(POST_CALL)
caller_epilogue
`push eax`
	`int`	1	0.00030

(RET) `or eax,` *literal* *callee_epilogue* (POST_CALL) *caller_epilogue* `mov [`reg_{bp}`], eax`	`int`	1	0.00030

(RET)
`mov eax,` *literal*
callee_epilogue
(POST_CALL)
caller_epilogue
`mov [`reg_{bp}`], eax`
(POST_CALL)
caller_epilogue
`push eax`
	`int`	1	0.00030

(RET)
`idiv ecx`
callee_epilogue
(POST_CALL)
caller_epilogue
`push eax`
	`int`	1	0.00025

(RET) `movsx eax, [`reg_{bp}`]` *callee_epilogue*	`int`	0.96	0.00118

A.5 Class `pointer`

Antecedents	Consequent	Confidence	Support
(RET) `mov eax,` *offset* *callee_epilogue*	`pointer`	1	0.00590
(RET) `add eax,` *offset* *callee_epilogue*	`pointer`	1	0.00359
(RET) *gen_mov_chain* `eax, [`reg_{bp}`],` *offset* *callee_epilogue*	`pointer`	1	0.00231
(RET) `lea eax, [`$reg_{bp}+reg_{ax}$`]` *callee_epilogue*	`pointer`	1	0.00207
(RET) `lea eax, [`$reg_{bp}+reg_{cx}$`]` *callee_epilogue*	`pointer`	1	0.00172
continues			

Antecedents	Consequent	Confidence	Support
(RET) *return_add_assign_int_to_ptr* `eax, [`reg_{bp}`], ecx, [`reg_{bp}`]` *callee_epilogue*	`pointer`	1	0.00157
(RET) `lea eax, [`$reg_{bp}+reg_{dx}$`]` *callee_epilogue*	`pointer`	1	0.00152

(RET)
`lea eax, [`reg_{bp}`]`
callee_epilogue
(POST_CALL)
caller_epilogue
`push eax`
	`pointer`	1	0.00148

(RET) *return_add_assign_int_to_ptr* `eax, [`reg_{bp}`], eax, [`reg_{bp}`]` *callee_epilogue*	`pointer`	1	0.00089
(RET) *add_int_to_ptr* `eax,` *offset* *callee_epilogue*	`pointer`	1	0.00064

(RET)
`mov eax, [`reg_{cx}`]`
callee_epilogue
(POST_CALL)
caller_epilogue
`mov` *address*`, eax`
	`pointer`	1	0.00030

(RET)
`mov eax, [`reg_{ax}`]`
callee_epilogue
(POST_CALL)
caller_epilogue
`mov` *address*`, eax`
	`pointer`	1	0.00030

(RET)
`mov eax, [`reg_{bp}`]`
callee_epilogue
(POST_CALL)
caller_epilogue
`mov` *address*`, eax`
(POST_CALL)
caller_epilogue
`push eax`
	`pointer`	1	0.00025

(RET) `lea eax, [`reg_{bp}`]` *callee_epilogue*	`pointer`	0.992	0.00585
(RET) `add eax,` *literal* *callee_epilogue* (POST_CALL) *caller_epilogue* `push eax`	`pointer`	0.962	0.00246
(RET) `mov eax,` *address* *callee_epilogue* (POST_CALL) *caller_epilogue* `push eax`	`pointer`	0.958	0.00113

A.6 Class `struct`

Antecedents	Consequent	Confidence	Support
(RET) `mov eax, [ebp+8]` *callee_epilogue*	`struct`	1	0.06321

A.7 Class `long long`

Antecedents	Consequent	Confidence	Support
(RET) `cdq` *callee_epilogue*	`long long`	1	0.04442
(RET) `mov edx,` [reg_{dx}] *callee_epilogue*	`long long`	1	0.00128

(POST_CALL)
caller_epilogue
`mov` *address*`, eax`
(POST_CALL)
caller_epilogue
`push edx`
	`long long`	1	0.00074

(RET) `mov edx,` [reg_{bp}+reg_{cx}] *callee_epilogue*	`long long`	1	0.00059
(RET) `mov edx,` *address* *callee_epilogue*	`long long`	1	0.00059
(POST_CALL) *caller_epilogue* `push edx`	`long long`	0.99	0.00984
(RET) `mov edx,` [reg_{cx}] *callee_epilogue*	`long long`	0.98	0.00236

A.8 Class `float`

Antecedents	Consequent	Confidence	Support
(RET) *assign_int_cast_to_float* [reg_{bp}], [reg_{bp}], [reg_{bp}] *callee_epilogue*	`float`	1	0.03807
(POST_CALL) *caller_epilogue* *fstp_dword* [reg_{sp}]	`float`	1	0.00836
(RET) *fld_dword* [reg_{bp}+reg_{dx}] *callee_epilogue*	`float`	1	0.00089
(RET) *fld_dword address* *callee_epilogue*	`float`	0.99	0.00954
continues			

Antecedents	Consequent	Confidence	Support
(RET) *fld_dword* [reg_{ax}] *callee_epilogue*	`float`	0.983	0.00280
(RET) *fld_dword* [reg_{dx}] *callee_epilogue*	`float`	0.952	0.00197

A.9 Class `double`

Antecedents	Consequent	Confidence	Support
(RET) *assign_int_cast_to_double* [reg_{bp}], [reg_{bp}], [reg_{bp}] *callee_epilogue*	`double`	1	0.03566
(RET) *fld_qword address* *callee_epilogue*	`double`	1	0.01033
(RET) *return_assign_double* [reg_{bp}], xmm0 *callee_epilogue*	`double`	1	0.00930
(RET) *fld_qword* [reg_{bp}] *callee_epilogue*	`double`	1	0.00266
(RET) *fld_qword* [reg_{dx}] *callee_epilogue*	`double`	1	0.00207
(RET) *fld_qword* [reg_{ax}] *callee_epilogue*	`double`	1	0.00192
(RET) *fld_qword* [reg_{cx}] *callee_epilogue*	`double`	1	0.00177
(RET) *fld_qword* [$reg_{bp}+reg_{ax}$] *callee_epilogue*	`double`	1	0.00074
(RET) *fld_qword* [$reg_{bp}+reg_{cx}$] *callee_epilogue*	`double`	1	0.00069

A.10 Class `void`

Antecedents	Consequent	Confidence	Support
(RET) mov [reg_{bp}], *literal* *callee_epilogue*	`void`	1	0.00349
(RET) mov *address*, *literal* *callee_epilogue*	`void`	1	0.00285
(RET) mov [reg_{ax}], ecx *callee_epilogue*	`void`	1	0.00182

Appendix B

Binary pattern generalizations

This appendix shows a complete list of the generalizations used in Chapter 5 to create the predictive models after applying the method in Chapter 4.

In the following table, *reg* variables represent registers, *literal* integer literals, *address* absolute addresses, **address* absolute addresses dereferences and *offset* relative addresses. Variables between normal-font brackets ([]) represent optional arguments, while typewriter-font brackets (`[]`) are the assembly brackets denoting register-based dereferences.

Assembly pattern	**Feature (Generalization)**

`pop` reg_1
...
`pop` reg_n
`pop ebp`
`ret`
 	
where: $reg_1, ..., reg_n \notin \{$`ebp`$\}$
$\sum_{i=1}^{n} \sum_{j=i}^{n} reg_i \neq reg_j$
	callee_epilog()

`pop` reg_1
...
`pop` reg_n
`mov esp, ebp`
`pop ebp`
`ret`
 	
where: $reg_1, ..., reg_n \notin \{$`ebp`$\}$
$\sum_{i=1}^{n} \sum_{j=i}^{n} reg_i \neq reg_j$
	callee_epilog()

`pop` reg_1
...
`pop` reg_n
`mov ecx, [ebp+var_4]`
`xor ecx, ebp`
`call @__security_check_cookie@4`
`mov esp, ebp`
`pop ebp`
`ret`
 	
where: $reg_1, ..., reg_n \notin \{$`ebp`$\}$
$\sum_{i=1}^{n} \sum_{j=i}^{n} reg_i \neq reg_j$
	callee_epilog()

continues	

Assembly pattern	Feature (Generalization)

pop reg_1
...
pop reg_n
mov esp, ebp
pop ebp
mov esp, ebx
pop ebx
ret
 	
where: reg_1, ..., $reg_n \notin$ {ebp, ebx}
$\sum_{i=1}^{n}\sum_{j=i}^{n} reg_i \neq reg_j$
	callee_epilog()

pop reg_1
...
pop reg_n
mov ecx, [ebp+var_4]
xor ecx, ebp
call @__security_check_cookie@4
mov esp, ebp
pop ebp
mov esp, ebx
pop ebx
ret
 	
where: reg_1, ..., $reg_n \notin$ {ebp, ebx}
$\sum_{i=1}^{n}\sum_{j=i}^{n} reg_i \neq reg_j$
	callee_epilog()

call *offset*	*caller_epilog*()
call *offset* add esp, *literal*	*caller_epilog*()

cond_jmp $offset_1$
mov [reg_1], $literal_1$
jmp $offset_2$
mov [reg_1], $literal_2$
 	
where: *cond_jmp* ∈ {jo, jno, js, jns, je, jz, jne, jnz, jb, jnae, jc,
↪ jnb, jae, jnc, jbe, jna, ja, jnbe, jl, jnge, jge, jnl, jle,
↪ jng, jg, jnle, jp, jpe, jnp, jpo, jcxz, jecxz}
$literal_1$, $literal_2$ ∈ {0, 1}
$literal_1 \neq literal_2$
$offset_1 \neq offset_2$
	bool_cast(reg_1)

mov arg_2, arg_1 *mov* arg_3, arg_2 ... *mov* arg_n, arg_{n-1} where: *mov* ∈ {mov, movzx, movsx} arg_1 ∈ {*reg*, [*reg*], **address*, *literal*} arg_2, ..., arg_n ∈ {*reg*, [*reg*], **address*}	*gen_mov_chain*(arg_n, ..., arg_1)

mov arg_2, arg_1
mov arg_3, arg_2
...
mov arg_n, arg_{n-1}
 	
where: arg_1 ∈ {*reg*, [*reg*], **address*, *literal*}
arg_2, ..., arg_n ∈ {*reg*, [*reg*], **address*}
	mov_chain(arg_n, ..., arg_1)

movss arg_2, arg_1
movss arg_3, arg_2
...
movss arg_n, arg_{n-1}
 	
where: arg_1, ..., arg_n ∈ {*reg*, [*reg*], **address*}
	mov_float_chain(arg_n, ..., arg_1)

continues

Assembly pattern	**Feature (Generalization)**
`movsd` arg_2, arg_1 `movsd` arg_3, arg_2 ... `movsd` arg_n, arg_{n-1} where: arg_1, ..., $arg_n \in$ {*reg*, `[`*reg*`]`, **address*}	*mov_double_chain*(arg_n, ..., arg_1)
bool_cast(reg_1) *mov_chain*(reg_{ax}, reg_n, ..., reg_1) *callee_epilogue* where: $reg_{ax} \in$ {`eax`, `ax`, `al`, `ah`}	*return_bool_cast*(reg_{ax}, reg_n, ..., reg_1)
mov_chain(reg_{ax}, reg_n, ..., reg_1, *literal*) *callee_epilogue* where: $reg_{ax} \in$ {`eax`, `ax`, `al`, `ah`} *literal* $\in$ {`0`, `1`}	*return_bool_literal*(reg_{ax}, reg_n, ..., reg_1)
gen_mov_chain(reg_2, reg_1) `add` reg_2, `1` *mov_chain*(reg_1, reg_2)	*inc_int*(reg_1)
gen_mov_chain(reg_2, reg_1) `sub` reg_2, `1` *mov_chain*(reg_1, reg_2)	*dec_int*(reg_1)
gen_mov_chain(reg_2, reg_1) `add` reg_2, *literal* *mov_chain*(reg_1, reg_2) where: *literal* > 1	*inc_ptr*(reg_1)
gen_mov_chain(reg_2, reg_1) `sub` reg_2, *literal* *mov_chain*(reg_1, reg_2) where: *literal* > 1	*dec_ptr*(reg_1)
`shl` reg_2, *literal* *mov_chain*(reg_1, **address*) `sub` reg_1, reg_2	*sub_int_to_ptr*(reg_1, **address*, reg_2)

`call __allmul`
mov_chain(*reg*, **address*)
`sub` *reg*, `eax`
	sub_int_to_ptr(*reg*, **address*)

`call __allmul` `add eax`, **address*	*add_int_to_ptr*(**address*)
sub_int_to_ptr(reg_1, **address*, [reg_2]) *mov_chain*(reg_{ax}, reg_1) *callee_epilogue* where: $reg_{ax} \in$ {`eax`, `ax`, `al`, `ah`}	*return_sub_int_to_ptr*(reg_{ax}, reg_1, ↪ **address*, [reg_2])
sub_int_to_ptr(reg_1, **address*, [reg_2]) *gen_mov_chain*(reg_{ax}, arg_1, reg_1) *callee_epilogue* where: $reg_{ax} \in$ {`eax`, `ax`, `al`, `ah`} $arg_1 \in$ {*reg*, `[`*reg*`]`, **address*}	*return_sub_assign_int_to_ptr*(reg_{ax}, arg_1, ↪ reg_1, **address*, [reg_2])
add_int_to_ptr(reg_1, **address*) *gen_mov_chain*(reg_{ax}, arg_1, reg_1) *callee_epilogue* where: $reg_{ax} \in$ {`eax`, `ax`, `al`, `ah`} $arg_1 \in$ {*reg*, `[`*reg*`]`, **address*}	*return_add_assign_int_to_ptr*(reg_{ax}, arg_1, ↪ reg_1, **address*)

continues

Assembly pattern	Feature (Generalization)
`lea` reg_1, *address gen_mov_chain(reg_{ax}, arg_1, reg_1) callee_epilogue where: $reg_{ax} \in$ {`eax`, `ax`, `al`, `ah`} $arg_1 \in$ {reg, [reg], *address}	return_add_assign_int_to_ptr(reg_{ax}, arg_1, ↪ reg_1, *address)
math_op arg_1 callee_epilogue where: math_op $\in$ {`div`, `idiv`} $arg_1 \in$ {reg, [reg], *address}	return_int_math_op(arg_1)
math_op arg_1 mov_chain(reg_{ax}, reg_{rem}) callee_epilogue where: math_op $\in$ {`div`, `idiv`} $arg_1 \in$ {reg, [reg], *address} $reg_{ax} \in$ {`eax`, `ax`, `al`} $reg_{rem} \in$ {`edx`, `dx`, `ah`}	return_int_math_op(reg_{ax}, arg_1)
math_op arg_1 arg_2 mov_chain(arg_{ax}, reg_1) callee_epilogue where: math_op $\in$ {`imul`, `sub`, `add`, `sar`, `sal`, `shr`, `shl`, `xor`, `or`, `and`} $arg_1 \in$ {reg, [reg], *address} $arg_2 \in$ {reg, [reg], *address, literal} $reg_{ax} \in$ {`eax`, `ax`, `al`, `ah`}	return_int_math_op(reg_{ax}, arg_1, arg_2)
math_op arg_2 mov_chain(reg_{ax}, arg_1) callee_epilogue where: math_op $\in$ {`div`, `idiv`} $arg_1 \in$ {reg, [reg], *address} $arg_2 \in$ {reg, [reg], *address} $reg_{ax} \in$ {`eax`, `ax`, `al`, `ah`}	return_int_math_op_assign(reg_{ax}, arg_1, ↪ arg_2)
math_op arg_2 mov_chain(reg_{ax}, arg_1, reg_{rem}) callee_epilogue where: math_op $\in$ {`div`, `idiv`} $arg_1 \in$ {reg, [reg], *address} $arg_2 \in$ {reg, [reg], *address} $reg_{rem} \in$ {`edx`, `dx`, `ah`}	return_int_math_op_assign(reg_{ax}, arg_1, ↪ arg_2)
math_op arg_2 arg_3 mov_chain(reg_{ax}, arg_1, arg_2) callee_epilogue where: math_op $\in$ {`imul`, `sub`, `add`, `sar`, `sal`, `shr`, `shl`, `xor`, `or`, `and`} $arg_1 \in$ {reg, [reg], *address} $arg_2 \in$ {reg, [reg], *address} $arg_3 \in$ {reg, [reg], *address, literal} $reg_{ax} \in$ {`eax`, `ax`, `al`, `ah`}	return_int_math_op_assign(reg_{ax}, arg_1, ↪ arg_2, arg_3)
`fstp` arg where: opcode(`fstp`)[0] = 0xD9 arg $\in$ {[reg], *address}	fstp_dword(arg)
`fstp` arg where: opcode(`fstp`)[0] = 0xDD arg $\in$ {[reg], *address}	fstp_qword(arg)

continues

Assembly pattern	**Feature (Generalization)**
`fld` *arg* where: opcode(`fld`)[0] = 0xD9 $arg \in \{$`[`*reg*`]`, **address*$\}$	*fld_dword*(*arg*)
`fld` *arg* where: opcode(`fld`)[0] = 0xDD $arg \in \{$`[`*reg*`]`, **address*$\}$	*fld_qword*(*arg*)
`fild` arg_1 *fstp_dword*(arg_2) *fld_dword*(arg_3) where: $arg_1 \in \{$`[`*reg*`]`, **address*$\}$ $arg_2 \in \{$`[`*reg*`]`, **address*$\}$ $arg_3 \in \{$`[`*reg*`]`, **address*$\}$	*assign_int_cast_to_float*(arg_3, arg_2, ↪ arg_1)
`fild` arg_1 *fstp_qword*(arg_2) *fld_qword*(arg_3) where: $arg_1 \in \{$`[`*reg*`]`, **address*$\}$ $arg_2 \in \{$`[`*reg*`]`, **address*$\}$ $arg_3 \in \{$`[`*reg*`]`, **address*$\}$	*assign_int_cast_to_double*(arg_3, arg_2, ↪ arg_1)
mov_float_chain(arg_n, ..., arg_1) *fld_dword*(arg_n) where: arg_1, ..., $arg_{n-1} \in \{$*reg*, `[`*reg*`]`, **address*$\}$ $arg_n \in \{$`[`*reg*`]`, **address*$\}$	*return_assign_float*(arg_n, ..., arg_1)
mov_double_chain(arg_n, ..., arg_1) *fld_qword*(arg_n) where: arg_1, ..., $arg_{n-1} \in \{$*reg*, `[`*reg*`]`, **address*$\}$ $arg_n \in \{$`[`*reg*`]`, **address*$\}$	*return_assign_double*(arg_n, ..., arg_1)
`mov al`, *literal* where: $literal > 1$	*assign_char_literal*()

Appendix C

Cnerator user manual

Cnerator is a C source code generation tool [70]. It can be used to generate large amounts of standard ANSI/ISO C source code, ready to be compiled by any standard language implementation. Cnerator is highly customizable to generate all the syntactic constructs of the C language, necessary to build accurate predictive models with machine learning algorithms.

C.1 Installation

You need a Python 3.7+ standard implementation. The only additional package to install is `numpy`:

```
pip install numpy
```

C.2 Usage

Then, you may just run Cnerator with no arguments to generate a random C program in the `out` directory:

```
python cnerator.py
```

There are plenty of options to customize Cnerator. To specify the probability of a particular language construct, you can use the `-p` or `--probs` option. The following command sets to 20% the probability of generating a function invocation when a new expression is required:

```
python cnerator.py -p "call_prob = {True: 0.2, False: 0.8}"
```

If more sophisticated probabilities are required, you can specify them in a JSON file and pass it as a parameter (see Section C.5 to know the JSON file format).

The following line passes an example JSON file in the `json/probabilities` folder where different probability distributions are specified for some syntactic constructs:

```
python cnerator.py -P json/probabilities/example_probs.json
```

Cnerator also allows the user to control the number and characteristics of all the functions to be generated. A JSON file is used for that purpose (Section C.6). The following command makes Cnerator generate one function for each high-level `return` type in the C programming language:

```
python cnerator.py -f json/functions/1-function-each-type.json
```

Sometimes, the user needs the output program to fulfill some requirements not guaranteed by the stochastic generation process. To this aim, Cnerator allows the specification of an ordered collection of Python post-process specification files. These post-processing files can modify the generated code to satisfy those requirements.

The following execution generates a random program and then executes two visitors: one to rename `func` functions to `proc` (and their invocations) when they return `void`; and another one to add a `__RETURN`*n*`__` label before each return statement:

```
python cnerator.py -V visitors.func_to_proc:visitors.return_instrumentation
```

To see all the options, just run the `-h` or `--help` options.

C.3 Command line arguments

These are the command line arguments provided by Cnerator (all of them are optional):

- `-o NAME` or `--output NAME`: `NAME` indicates the output file name, without the file extension. The default value is `main`.
- `-O PATH` or `--output_dir PATH`: `PATH` is the output directory where the C source code is placed. If the directory does not exist, it is created. The default value is `out`.
- `-n NUMBER` or `--nfiles NUMBER`: Generates the output program in `NUMBER` compilation units. Each compilation unit is a pair of `.c` and `.h` files. A compilation unit can be compiled independently, even though they commonly depend on other compilation units. The default value is 2.
- `-r RECURSION_LIMIT` or `--recursion RECURSION_LIMIT`: Defines the maximum number of Python recursive invocations. This parameter may be modified when massive codebases are being modified, checking that the runtime environment provides sufficient memory. The default value is 50000.
- `-v` or `--verbose`: Enables the verbose mode to show runtime messages. By default, the verbose mode is disabled.
- `-d` or `--debug`: Generates debug information, comprising call graphs and struct structures in `.dot` files. By default, the debug option is disabled.
- `-p PROBS` or `--probs PROBS`: `PROBS` represents a semicolon-separated list of probabilities and their values for different syntactic constructs. An example

use is `"call_prob={True:0.2,False:0.8}; param_number_prob={0:0.2, ↪ 1:0.3,2:0.3,3:0.2}"`.

- `-P PROBSFILE` or `--probsfile PROBSFILE`: `PROBSFILE` is a JSON file specifying some probabilities for different language constructs. Different examples are provided in the `json/probabilities` directory. The structure of `PROBSFILE` JSON files is described in Section C.5.
- `-f FUNCTIONS` or `--functions FUNCTIONS`: `FUNCTIONS` is a JSON file specifying conditions of the functions to be created by Cnerator. The user can express properties that the generated functions will fulfill. Different example files are provided in the `json/functions` directory.
- `-V VISITORS` or `--visitors VISITORS`: An ordered colon-separated list of visitors to adapt, process or modify the program representation generated by Cnerator. Once the visitors are run, Cnerator takes the final program representation and generates the final C source code. An example value of `VISITORS` is `visitors.func_to_proc:visitors.return_instrumentation`. The `visitors` directory provides different examples of visitor implementations. A brief description of how to implement of such visitors is presented in Section C.7.
- `-h` or `--help`: Shows a description of the command line arguments, including the default values.

C.4 Syntactic constructs

As mentioned, Cnerator allows defining the probabilities of different syntactic constructs of the C programming language. What follows is a description of all the constructs and their unique identifiers, when the `-p` or `-P` options are used:

- `primitive_types_prob`: probabilities among primitive types (default: equal probability for all the types).
- `assignment_types_prob`: assignment type (default: equal probability for all the types).
- `augmented_assignment_types_prob`: augmented assignment type (`+=`, `-=`, `*=`, etc.) (default: equal probability for primitive types).
- `all_types_prob`: type probability when any type may occur in a syntax construction (default: equal probability for any type).
- `array_size`: size of the arrays to be created (default: 1-10).
- `reuse_struct_prob`: when a struct is needed, probability of using and existing one rather than creating a new one (default: 90%).
- `enhance_existing_struct_prob`: when a struct is needed, probability of extending an existing one with the demanded field rather than creating a new one (default: 70%).

- `array_literal_initialization_prob`: array initialization upon definition (default: 10%).
- `struct_literal_initialization_prob`: struct initialization upon definition (default: 10%).
- `exp_depth_prob`: expression depth (default: equal probabilities for [0-2]).
- `return_exp_depth_prob`: expression depth for the particular expressions to be returned by functions (default: equal probabilities for [0-2]).
- `call_prob`: probability that a new expression is a function invocation (default: 20%).
- `basic_expression_prob`: basic expressions (default: same probability among literal, local variable, global variable, and param variable).
- `param_number_prob`: number of parameters (default: 10% for 1, 20% for [1,4] and 5% for [5,6]).
- `param_types_prob`: types of the parameters (default: all types are equally likely).
- `stmt_invocation_prob`: invocation statements to functions or procedures (default: function=88%, procedure=12%).
- `return_types_prob`: function return types (default: all types are equally likely).
- `int_emulate_bool`: probability of generating a `bool` return (0 or 1) when an `int` type is expected (default: 20%).
- `new_global_var_prob`: probability of creating a new global variable when one of the expected types already exists (default: 1%).
- `new_local_var_prob`: probability of creating a new local variable when one of the expected types already exists (default: 1%).
- `reuse_func_prob`: probability of reusing an existing function of the expected type (default: 99%).
- `reuse_proc_prob`: probability of reusing an existing procedure of the expected type (default: 99%).
- `global_or_local_as_basic_lvalue_prob`: When a basic l-value needs to be generated, this is the probability of using a global variable; otherwise, a local variable is used (default: 50%).
- `basic_or_compound_stmt_prob`: probability of generating a basic (no block) or compound statement (default: basic=70%, compound=30%).
- `function_basic_stmt_prob`: each kind of basic (no block) statement in functions (default: assignment=60%, invocation=20%, increment / decrement=20%, augmented assignment=10%).

- `function_compound_stmt_prob`: each kind of compound statement (with blocks) in functions (equal probability for `Block`, `If`, `Switch`, `Do, While`, `For`).
- `stmt_depth_prob`: statement depth (default: equal probabilities for [0-2]).
- `procedure_basic_stmt_prob`: each kind of basic (no block) statement in functions (default: assignment=60%, invocation=20%, increment / decrement=20%, augmented assignment=10%).
- `procedure_compound_stmt_prob`: each kind of compound statement (with blocks) in functions (equal probability for `Block`, `If`, `Switch`, `Do, While`, `For`).
- `number_stmts_main_prob`: number of statements in the main function (default: equal probabilities between 5 and 10).
- `number_stmts_func_prob`: number of statements in functions (default: 20% for [1,4] and 10% for [5, 6]).
- `number_stmts_block`: number of statements in blocks (default: 1/3 for 1, 2 and 3).
- `else_body_prob`: probability of generating an `else` body for an `if` statement (default: 50%).
- `number_cases_prob`: number of cases in `switch` statements (default: equal probabilities between 1 and 4).
- `cases_type_prob`: type of the cases clauses in `switch` statements (default: equal probabilities between types promotable to `int`, excluding `bool`).
- `default_switch_prob`: probability of generating a default case in a `switch` statement (default: 80%).
- `break_case_prob`: probability of having a `break` statement at the end of a `case` block (default: 70%).
- `return_at_end_if_else_bodies_prob`: probability of having a `return` at the end of an `if` / `else` blocks (default: 15%).
- `return_at_end_case_prob`: probability of having a `return` at the end of the cases clauses in a `switch` statement (default: 15%).
- `implicit_promotion_bool`: if an expression is expected, the probability to generate it with another type promotable to the expected one (default: 30%).
- `promotions_prob`: promotions between types (default: all the conversions are equally likely).

C.5 Probability specification files

Probability specification files are JSON documents that the user can use to define the probability of different syntactic constructs (Section C.4). The following JSON file shows an example:

```
{
   "function_basic_stmt_prob": {
      "assignment": 0.3,
      "invocation": 0.4,
      "augmented_assignment": 0.15,
      "incdec": 0.1,
      "expression_stmt": 0.05
   },
   "array_literal_initialization_prob": {
      "True": 0.1, "False": 0.9
   },
   "primitive_types_prob": {
      "__prob_distribution__": "equal_prob",
      "__values__": [
         "ast.Bool",
         "ast.SignedChar",
         "ast.UnsignedChar",
         "ast.SignedShortInt"
      ]
   },
   "param_number_prob": {
      "__prob_distribution__": "proportional_prob",
      "__values__": {
         "0": 1, "1": 2, "2": 3,
         "3": 3, "4": 2, "5": 1
      }
   },
   "number_stmts_main_prob": {
      "__prob_distribution__": "normal_prob",
      "__mean__": 10,
      "__stdev__": 3
   }
}
```

The first entry (`function_basic_stmt_prob`) defines the probability of building basic statements (*i.e.*, statements not containing other statements, unlike `for` and `switch`), and the second one (`array_literal_initialization_prob`) states when an array definition should initialize their values. These two examples specify fixed probabilities that must sum zero. The three remaining entries describe uniform/equal, proportional and normal distributions to define, respectively, the usage of primitive types (`primitive_types_prob`), the number of function parameters (`param_number_prob`) and statements in the `main` function (`number_stmts_main_prob`).

C.6 Function generation files

Cnerator provides the capability of specifying features to be fulfilled by the generated functions. The following JSON file shows an example use of such capacity:

```
{
  "function_returning_void": {
    "total": 1000,
    "condition": "lambda f: isinstance(f.return_type, ast.Void)"
  },
  "function_returning_bool": {
    "total": 1000,
    "condition": "lambda f: isinstance(f.return_type, ast.Bool)"
  },
  "function_with_if_else": {
    "total": 1,
    "condition": "lambda f: any(stmt for stmt in f.children
     if isinstance(stmt, ast.If) and any(stmt.else_statements))"
  }
}
```

The first two entries (`function_returning_void` and `function_returning_bool`) enforce Cnerator to generate 1000 functions returning `void` and the same number of functions returning `bool`. The only condition in the `lambda` expressions checks that the returned type is the expected one. The last entry (`function_with_if_else`) shows a different example, where the user demands Cnerator to generate a function containing an `if` statement with an `else` clause.

C.7 Post-processing specification files

Cnerator provides a mechanism to process/modify program representation before the final source code generalization. The following code shows an example Python post-process specification file:

```
from functools import singledispatch
from cnerator import ast

def _instrument_statements(statements: List[ast.ASTNode]) -> List[ast.ASTNode]:
  """Includes a unique label before any return statement"""
  instrumented_stmts = []
  for stmt in statements:                 # iterates through the statements
    if isinstance(stmt, ast.Return):      # if the statement is return...
      label = ast.Label(generate_label()) # creates new AST node for the label
      instrumented_stmts.append(label)    # and places the label before return
      visit(stmt)                         # traverses the statement
      instrumented_stmts.append(stmt)     # appends return after the label
  return instrumented_stmts

@visit.register(ast.Function)
def _(function: ast.Function):
  """Traverses a function definition to add a unique label before
         each return statement"""
  function.stmts = _instrument_statements(function.stmts)

@visit.register(ast.Do)
@visit.register(ast.While)
@visit.register(ast.For)
@visit.register(ast.Block)
def _(node):
  """Traverses a control flow statement to add a unique
```

```
          label before each return statement"""
  node.statements = _instrument_statements(node.statements)

_return_label_counter = 0

def generate_label() -> str:
  """Generates a new unique label string"""
  global _return_label_counter
  _return_label_counter += 1
  return f"__RETURN{_return_label_counter}__"

...
```

The previous code traverses the representation of the generated program (*i.e.*, its Abstract Syntax Tree (AST)), and adds a unique label before each `return` statement. The `_instrument_statements function` takes a list of statements (represented as AST nodes) and adds a unique label—prefixed with `__RETURN`—before each `return` statement. This is later used in the traversal of function definitions (`ast.Function`), and `do`, `while`, `for` and block statements—`if` and `switch` control flow statements follow the same template. This instrumentation technique was used to associate fragments of binary code with their corresponding high-level `return` statements.

The previous code is an instance of an introspective implementation of the Visitor design pattern. The `visit` annotations indicate the AST node to be traversed.

Appendix D

Hyperparameters

In Chapter 5 we detail how to create predictive models for the decompilation of function return types. We use different machine-learning algorithms implemented by scikit-learn. For each algorithm, we tune the hyperparameters of each model. To that end, we use `GridSearchCV`, which performs an exhaustive search over the specified hyperparameter values. The 80% training set is used to validate the hyperparameters with 3-fold stratified cross-validation (`StratifiedShuffleSplit`).

These are the hyperparameters used for each algorithm:

- AdaBoost: `n_estimators` = `500`, `base_estimator` = `DecisionTreeClassifier()`, `learning_rate` = `1.2`, `algorithm` = `"SAMME"`.
- Bernoulli naïve Bayes: `binarize` = `None`, `alpha` = `0.2`, `fit_prior` = `True`.
- Decision tree: `max_features` = `sqrt`, `min_samples_split` = `7`, `criterion` = `gini`, `splitter` = `best`.
- Extremely randomized trees: `warm_start` = `True`, `n_jobs` = `-1`, `n_estimators` = `100`, `min_samples_split` = `10`, `criterion` = `gini`, `max_features` = `log2`.
- Gradient boosting: `warm_start` = `True`, `loss` = `deviance`, `max_leaf_nodes` = `None`, `learning_rate` = `0.2`, `min_samples_leaf` = `1`, `n_estimators` = `100`, `subsample` = `1`, `min_weight_fraction_leaf` = `0.0`, `criterion` = `friedman_mse`, `min_impurity_split` = `1e-07`, `max_features` = `sqrt`, `min_samples_split` = `7`, `max_depth` = `5`.
- K-nearest neighbors: `n_neighbors` = `8`, `metric` = `manhattan`, `n_jobs` = `-1`, `weights` = `distance`, `algorithm` = `brute`.
- Linear support vector machine: `loss` = `squared_hinge`, `C` = `1`, `tol` = `1`, `penalty` = `l2`, `multi_class` = `crammer_singer`, `dual` = `True`, `class_weight` = `balanced`.
- Logistic regression: `penalty` = `l2`, `C` = `25`, `tol` = `0.0001`, `dual` = `False`, `solver` = `liblinear`.

– Multilayer perceptron: `shuffle` = `True`, `solver` = `adam`, `activation` = `logistic`, `hidden_layer_sizes` = `(100, )`, `tol` = `0.0001`, `alpha` = `0.01`.

– Multinomial naïve Bayes: `alpha` = `0.35`, `fit_prior` = `True`.

– Perceptron: `penalty` = `l1`, `alpha` = `1e-09`, `n_jobs` = `-1`, `eta0` = `10.0`, `n_iter` = `11`.

– Random forest: `warm_start` = `True`, `n_jobs` = `-1`, `n_estimators` = `100`, `min_samples_split` = `8`, `criterion` = `gini`, `max_features` = `log2`.

– Support vector machine: `kernel` = `sigmoid`, `C` = `100.0`, `probability` = `False`, `shrinking` = `True`, `decision_function_shape` = `ovo`, `tol` = `0.1`, `cache_size` = `1024`, `coef0` = `0.0`, `gamma` = `0.01`, `class_weight` = `balanced`.

Appendix E

Publications

The research work of this PhD dissertation has been submitted for publication in the following journals:

- Javier Escalada, Francisco Ortin, Ted Scully. An Efficient Platform for the Automatic Extraction of Patterns in Native Code. *Scientific Programming*, volume 2017, pp. 1-17, March 2017, doi: 10.1155/2017/3273891, JCR Impact Factor 1.344 (Q2).
- Javier Escalada, Ted Scully, Francisco Ortin. Improving type information inferred by decompilers with supervised machine learning. *Expert systems with applications* (article under review), arXiv:2101.08116, JCR Impact Factor 5.451 (Q1).
- Francisco Ortin, Javier Escalada. Cnerator: a Python application for the controlled stochastic generation of standard C source code. *SoftwareX* (article under 2nd review – minor changes), JCR Impact Factor 0.945 (Q4)[1].
- Francisco Ortin, Javier Escalada, Oscar Rodriguez-Prieto. Big Code: New Opportunities for Improving Software Construction. *Journal of Software*, volume 11, issue 11, pp. 1083-1088, November 2016, doi: 10.17706/jsw.11.-11.1083-1088, SciMago Impact Factor 0.708 (Q3).
- Javier Escalada, Francisco Ortin. An Adaptable Infrastructure to Generate Training Datasets for Decompilation issues. *Advances in Intelligent Systems and Computing (New Perspectives in Information Systems and Technologies)*, pages 85–94, 2014, doi: 10.1007/978-3-319-05948-8_9, SciMago Impact Factor 0.634 (Q3). Selected paper from the World Conference on Information Systems and Technologies (WorldCIST), Madeira (Portugal), April 2014, Core C.

[1] This journal has already been indexed in the Journal Citation Reports, but its first impact factor will be published in May 2021. This information is available at `https://mjl.clarivate.com/home`

References

[1] R. Nigel Horspool and Nenad Marovac. An Approach to the Problem of Detranslation of Computer Programs. *The Computer Journal*, 23(3):223–229, aug 1980. 1, 29

[2] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, School of Computing Science, Queensland University of Technology, AU, 1994. 1, 9, 44

[3] Michael James Van Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, School of Information Technology and Electrical Engineering, University of Queensland, AU, 2007. 1, 9, 44

[4] Francisco Ortin, Javier Escalada, and Oscar Rodriguez-Prieto. Big Code: New Opportunities for Improving Software Construction. *Journal of Software*, 11(11):1083–1008, nov 2016. 1

[5] DARPA. MUSE Envisions Mining “Big Code” to Improve Software Reliability and Construction. https://www.darpa.mil/news-events/2014-03-06a, 2021. 1

[6] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting Program Properties from ”Big Code”. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL ’15*, volume 50, pages 111–124, New York, New York, USA, jan 2015. ACM Press. 1, 71

[7] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-Based Statistical Translation of Programming Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software - Onward! ’14*, pages 173–184, New York, New York, USA, oct 2014. ACM Press. 1

[8] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, may 2014. 1

[9] Francisco Ortin, Oscar Rodriguez-Prieto, Nicolas Pascual, and Miguel Garcia. Heterogeneous tree structure classification to label java programmers according to their expertise level. *Future Generation Computer Systems*, 105:380 – 394, 2020. 1, 31

[10] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural Nets Can Learn Function Type Signatures From Binaries. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 99–116, Vancouver, BC, aug 2017. USENIX Association. 2, 6, 35, 71

[11] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting Debug Information in Stripped Binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680, New York, NY, USA, jan 2018. ACM. 2, 6

[12] Deborah S. Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings*, volume 2018-March, pages 346–356. Institute of Electrical and Electronics Engineers Inc., apr 2018. 2, 7, 71

[13] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving Exact Decompilation. In *Proceedings 2018 Workshop on Binary Analysis Research*, Reston, VA, jul 2018. Internet Society. 2, 8

[14] ISO/IEC. ISO/IEC 9899:2018 - Information technology — Programming languages — C. `https://www.iso.org/standard/74528.html`, 2021. 4, 62

[15] David Everett Rumelhart, Geoffrey Everest Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. 6, 71

[16] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3(null):1137–1155, March 2003. 6

[17] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *In Proc. 18th International Conf. on Machine Learning 18th International Conf. on Machine Learning*, ICML '01, pages 282–289, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. 6, 18, 20

[18] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 463–469, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. 6, 10

[19] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, ESOP '99, page 208–223, Berlin, Heidelberg, 1999. Springer-Verlag. 6, 51, 73

[20] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978. 7

[21] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011. 7, 10, 73

[22] Gogul Balakrishnan. *WYSINWYX: WHAT YOU SEE IS NOT WHAT YOU EXECUTE*. PhD thesis, Computer Science Deptartement, University of Wisconsin-Madison, 2007. 7, 73

[23] Juan Caballero and Zhiqiang Lin. Type Inference on Executables. *ACM Computing Surveys*, 48(4):1–35, may 2016. 7, 29, 51, 73

[24] Easwaran Raman and David Isaac August. Recursive data structure profiling. In *Proceedings of the 2005 Workshop on Memory System Performance*, MSP '05, page 5–14, New York, NY, USA, 2005. Association for Computing Machinery. 7

[25] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. Binary Code Extraction and Interface Identification for Security Applications. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2010. 7

[26] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. Towards Neural Decompilation. *ArXiv*, 2019. 7

[27] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian Facebook, Farinaz Koushanfar, Jishen Zhao, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An End-to-End Neural Program Decompiler. In H Wallach, H Larochelle, A Beygelzimer, F d'Alché-Buc, E Fox, and R Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 3708–3719. Curran Associates, Inc., 2019. 7, 8

[28] Nal Kalchbrenner and Phil Blunsom. Recurrent continuous translation models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1700–1709, Seattle, Washington, USA, October 2013. Association for Computational Linguistics. 7

[29] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. *Advances in Neural Information Processing Systems*, 4(January):3104–3112, sep 2014. 7

[30] Matthew Henderson, Blaise Thomson, and Steve J. Young. Robust dialog state tracking using delexicalised recurrent neural networks and unsupervised adaptation. *2014 IEEE Spoken Language Technology Workshop (SLT)*, pages 360–365, 2014. 8

[31] Hongfa Xue, Shaowen Sun, Guru Venkataramani, and Tian Lan. Machine Learning-Based Analysis of Program Binaries: A Comprehensive Study. *IEEE Access*, 7:65889–65912, 2019. 8

[32] Nathan NE Rosenblum, Xiaojin Zhu, BP Barton Miller, and Karen Hunt. Learning to analyze binary computer code. In *Proceedings of the 23rd Conference on Artificial Intelligence*, pages 798–804, Chicago, 2008. 8, 12, 18, 19

[33] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Security Symposium*, pages 845–860, San Diego, CA, aug 2014. USENIX Association. 8, 11, 18

[34] Rene De La Briandais. File searching using variable length keys. In *Papers Presented at the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), page 295–298, New York, NY, USA, 1959. Association for Computing Machinery. 8

[35] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing Functions in Binaries with Neural Networks. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 611–626, Washington, D.C., aug 2015. USENIX Association. 8

[36] Nathan E. Rosenblum, Barton P. Miller, and Xiaojin Zhu. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '10*, PASTE '10, pages 21–28, Toronto, Ontario, Canada, 2010. ACM Press. 8, 12, 31, 60, 70

[37] Nathan Rosenblum, Barton P. Miller, and Xiaojin Zhu. Recovering the toolchain provenance of binary code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*, ISSTA '11, pages 100–110, Toronto, Ontario, Canada, jul 2011. ACM Press. 9, 12, 31, 70

[38] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of Machine Learning Techniques for Malware Analysis. *Computers & Security*, 81:123–147, oct 2017. 9

[39] Mamoun Alazab, Sitalakshmi Venkatraman, Paul Watters, and Moutaz Alazab. Zero-Day Malware Detection Based on Supervised Learning Algorithms of API Call Signatures. In *Proceedings of the Ninth Australasian Data Mining Conference - Volume 121*, AusDM '11, page 171–182, AUS, 2011. Australian Computer Society, Inc. 9

[40] Hemant Rathore, Swati Agarwal, Sanjay K. Sahay, and Mohit Sewak. Malware detection using machine learning and deep learning. *Lecture Notes in Computer Science*, page 402–411, 2018. 9

[41] Cristina Cifuentes. The DCC decompiler (through Wayback Machine). https://web.archive.org/web/20131209235003/http://itee.uq.edu.au/~cristina/dcc.html, 2021. 9

[42] Satish Kumar. DisC - Decompiler for TurboC. https://www.debugmode.com/dcompile/disc.htm, 2021. 9, 44

[43] Michael James Van Emmerik. Boomerang Decompiler. http://boomerang.sourceforge.net/, 2021. 9

[44] Francois Chagnon. EiNSTeiN-/decompiler. https://github.com/EiNSTeiN-/decompiler, 2021. 9

[45] Giampiero Caprino. REC Decompiler. http://www.backerstreet.com/rec/rec.htm, 2021. 9, 44

[46] Hex-Rays. Hex Rays Decompiler. https://www.hex-rays.com/products/decompiler/, 2021. 9, 44

[47] Ilfak Guilfanov. Simple type system for program reengineering. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 357–361. IEEE Comput. Soc, 2001. 9, 44

[48] Ilfak Guilfanov. Decompilers and beyond. In *Black Hat USA*, 2008. 9, 44

[49] Hex-Rays. Fast Library Identification and Recognition Technology. https://www.hex-rays.com/products/decompiler/, 2021. 10

[50] Katerina Troshina, Alexander Chernov, and Yegor Derevenets. C Decompilation: Is It Possible? In Mikhail A Bulyonkov and Robert Glück, editors, *Proceedings of International Workshop on Program Understanding*, pages 18–27, Altai Mountains, Russia, 2009. Ershov Institute of Informatics Systems, Siberian Branch of the Russian Academy of Sciences. 10, 44, 71

[51] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. SmartDec: Approaching C++ Decompilation. In *2011 18th Working Conference on Reverse Engineering*, pages 347–356. IEEE, oct 2011. 10, 44, 71

[52] Yegor Danilov. GitHub yegord/snowman. https://github.com/yegord/snowman, 2021. 10, 44, 72

[53] Cryptic Apps EURL. Hopper Disassembler. https://www.hopperapp.com/, 2021. 10, 44

[54] Edward J. Schwartz, Maverick Woo, David Brumley, and Jonghyup Lee. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *USENIX Security Symposium*, pages 353–368, Washington, D.C., 2013. USENIX. 10, 44

[55] Avast Software. GitHub avast/retdec. https://github.com/avast/retdec, 2021. 10, 44, 72

[56] Jakub Křoustek. *Retargetable Analysis of Machine Code*. PhD thesis, Faculty of Information Technology, Brno University of Technology, CZ, 2015. 10, 44

[57] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004. 10

[58] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Proceedings 2015 Network and Distributed System Security Symposium*, Reston, VA, 2015. Internet Society. 10, 44

[59] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, pages 158–177. Institute of Electrical and Electronics Engineers Inc., aug 2016. 10, 44

[60] Javier Escalada and Francisco Ortin. An adaptable infrastructure to generate training datasets for decompilation issues. In Álvaro Rocha, Ana Maria Correia, Felix . B Tan, and Karl . A Stroetmann, editors, *New Perspectives in Information Systems and Technologies, Volume 2*, pages 85–94, Cham, 2014. Springer International Publishing. 11

[61] Javier Escalada, Francisco Ortin, and Ted Scully. An Efficient Platform for the Automatic Extraction of Patterns in Native Code. *Scientific Programming*, 2017:1–16, 2017. 11

[62] Igor Santos, Yoseba K. Penya, Jaime Devesa, and Pablo García Bringas. N-grams-based file signatures for malware detection. In *Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS), Volume AIDSS*, pages 317–320, 2009. 12

[63] Ohm Sornil Chatchai Liangboonprakong. Classification of malware families based on n-grams sequential pattern features. In *8th IEEE Conference on Industrial Electronics and Applications*, ICIEA'13, pages 777–782, 2013. 12

[64] LLVM. clang: a C language family frontend for LLVM. `http://clang.llvm.org`, 2021. 20

[65] Elias Bachaalany. GitHub: IDAPython. `https://github.com/idapython`, 2021. 20, 72

[66] David Beazley. Understanding the Python GIL. In *PyCON Python Conference*, 2010. 21

[67] Dusty Phillips. *Python 3 Object-Oriented Programming - Second Edition*. Packt Publishing Ltd, Livery Place, Birmingham, UK, 2 edition, 2015. 21

[68] Python Software Foundation. multiprocessing — Process-based "threading" interface. https://docs.python.org/2/library/multiprocessing.html, 2021. 21

[69] Jose Manuel Redondo, Francisco Ortin, and Juan Manuel Cueva Lovelle. Optimizing reflective primitives of dynamic languages. *International Journal of Software Engineering and Knowledge Engineering*, 18(6):759–783, 2008. 22

[70] Francisco Ortin and Javier Escalada. Cnerator: a Python application for the controlled stochastic generation of standard C source code. *SoftwareX (article under 2nd review – minor changes)*, 2021. 22, 87

[71] Francisco Ortin, Luis Vinuesa, and Jose Manuel Felix. The DSAW aspect-oriented software development platform. *International Journal of Software Engineering and Knowledge Engineering*, 21(7):891–929, 2011. 22

[72] Gene Myron Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. 23, 24

[73] Javier Escalada and Francisco Ortin. Platform Implementation. www.reflection.uniovi.es/decompilation/download/2016/sp, 2021. 28

[74] Steven Stanley Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. 30, 60

[75] David Watt and Deryck Brown. *Programming Language Processors in Java: Compilers and Interpreters*. Pearson education. Prentice Hall, 2000. 30

[76] Gregory Conti, Erik Dean, Matthew Sinda, and Benjamin Sangster. Visual reverse engineering of binary and data files. In John R. Goodall, Gregory Conti, and Kwan-Liu Ma, editors, *Visualization for Computer Security*, pages 1–17, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. 31, 70

[77] Emily R. Jacobson, Nathan Rosenblum, and Barton P. Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools - PASTE '11*, PASTE'11, pages 1–8, Szeged, Hungary, 2011. ACM Press. 31, 70

[78] Yang Li and Tao Yang. *Word Embedding for Understanding Natural Language: A Survey*, pages 83–104. Springer International Publishing, Cham, 2018. 32

[79] Lyan Verwimp, Joris Pelemans, Hugo Van hamme, and Patrick Wambacq. Character-word LSTM language models. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 417–427, Valencia, Spain, April 2017. Association for Computational Linguistics. 32

[80] Lior Rokach and Oded Maimon. Top-down induction of decision trees classifiers - a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 35(4):476–487, 2005. 33

[81] Cristina Cifuentes and Kevin John Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995. 35

[82] Jerome Harold Friedman and Bogdan E. Popescu. Predictive learning via rule ensembles. *The Annals of Applied Statistics*, 2(3):916 – 954, 2008. 35

[83] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, May 2000. 35

[84] Javier Escalada, Ted Scully, and Francisco Ortin. Improving type information inferred by decompilers with supervised machine learning. *Expert systems with applications (article under review)*, 2021. 37

[85] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011. 42

[86] Ron Kohavi and George H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, dec 1997. 43

[87] Hex-Rays. Hex Rays IDA. `https://www.hex-rays.com/products/ida/`, 2021. 44

[88] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, October 2007. 45, 50

[89] Yiming Yang and Xin Liu. A re-examination of text categorization methods. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '99, page 42–49, New York, NY, USA, 1999. Association for Computing Machinery. 45

[90] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427 – 437, 2009. 45

[91] Juri Opitz and Sebastian Burst. Macro F1 and macro F1. *ArXiv*, 2019. 45

[92] Brian Wilson. Kernighan and Dennis MacAlistair Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988. 51

[93] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. 56, 74

[94] Microsoft. Calling Conventions — Microsoft Docs. https://docs.microsoft.com/en-us/cpp/cpp/calling-conventions, 2021. 59

[95] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51(4), July 2018. 62

[96] Hlib Babii, Andrea Janes, and Romain Robbes. Modeling vocabulary for big code machine learning, 2019. 62

[97] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery. 62

[98] Francisco Ortin, Jose Quiroga, Jose Manuel Redondo, and Miguel Garcia. Attaining multiple dispatch in widespread object-oriented languages. *Dyna*, 81(186):242–250, 2014. 64, 67

[99] Lukasz Langa. singledispatch Python package 3.4.0.3. https://pypi.org/project/singledispatch/, 2021. 64

[100] Oscar Rodriguez-Prieto, Alan Mycroft, and Francisco Ortin. An efficient and scalable platform for Java source code analysis using overlaid graph representations. *IEEE Access*, 8:72239–72260, 2020. 64

[101] Gamma Erich, Helm Richard, Johnson Ralph, and Vlissides John. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional Computing Series, 1995. 64

[102] Francisco Ortin, Benjamin López, and J. Baltasar García Pérez-Schofield. Separating adaptable persistence attributes through computational reflection. *IEEE Software*, 21(6):41–49, 2004. 67

[103] Mark Vicent Yason Paul Vicent Sabanal. BlackHat DC '07 – Reversing C++. https://www.blackhat.com/presentations/bh-dc-07/Sabanal_Yason/Paper/bh-dc-07-Sabanal_Yason-WP.pdf, 2021. 71

[104] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics. 71

[105] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, Jan 2009. 72

[106] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, Jan 2021. 72

[107] Hex-Rays. Hex Rays IDA SDK. https://www.hex-rays.com/products/ida/tech/plugin/, 2021. 72

[108] Stephen Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, Feb 1991. 73

[109] Qiang Zeng, Jignesh M. Patel, and David Page. Quickfoil: Scalable inductive logic programming. *Proceedings of the VLDB Endowment*, 8(3):197–208, 2014. 73