Hindawi
Scientific Programming
Volume 2017, Article ID 3273891, 16 pages
https://doi.org/10.1155/2017/3273891

Research Article

An Efficient Platform for the Automatic Extraction of Patterns in Native Code

Javier Escalada,[1] Francisco Ortin,[1] and Ted Scully[2]

[1]*Computer Science Department, University of Oviedo, Calvo Sotelo s/n, 33007 Oviedo, Spain*
[2]*Cork Institute of Technology, Computer Science Department, Rossa Avenue, Bishopstown, Cork, Ireland*

Correspondence should be addressed to Francisco Ortin; ortin@uniovi.es

Received 30 September 2016; Revised 26 December 2016; Accepted 17 January 2017; Published 28 February 2017

Academic Editor: Raphaël Couturier

Different software tools, such as decompilers, code quality analyzers, recognizers of packed executable files, authorship analyzers, and malware detectors, search for patterns in binary code. The use of machine learning algorithms, trained with programs taken from the huge number of applications in the existing open source code repositories, allows finding patterns not detected with the manual approach. To this end, we have created a versatile platform for the automatic extraction of patterns from native code, capable of processing big binary files. Its implementation has been parallelized, providing important runtime performance benefits for multicore architectures. Compared to the single-processor execution, the average performance improvement obtained with the best configuration is 3.5 factors over the maximum theoretical gain of 4 factors.

1. Introduction

Many software tools analyze programs, looking for specific patterns defined beforehand. When a pattern is found, an action is then performed by the tool (e.g., improve the quality, security, or performance of the input program). Patterns are defined for both high-level and binary code. Different examples of tools that analyze high-level patterns are refactoring tools, code quality analyzers, or detectors of common programming mistakes. In the case of binary code, examples are decompilers, packed executable file recognizers, authorship analyzers, or malware detectors.

In the traditional approach, an expert identifies those patterns to be found by the software tool. His or her expertise is used to define the code patterns that should be searched (e.g., for improving the application). On the contrary, a machine learning approach can be used to build models, which can then be applied to large repositories of code to effectively search for and identify specific patterns. This second approach allows the analysis of huge amounts of programs, sometimes detecting patterns not found with the traditional approach. Additionally, this approach could automatically evaluate the accuracy of the patterns obtained.

An emerging research topic called *big code* has recently appeared [1]. Big code is based on the idea that open source code repositories (e.g., GitHub, SourceForge, BitBucket, and CodePlex) can be used to create new kinds of programming tools and services to improve software reliability and construction, using machine learning and probabilistic reasoning. One of the research lines included in big code is finding extrapolated patterns, detecting software anomalies, or computing the cooccurrence of different patterns in the same program [2].

One example (more real examples can be consulted in [2]) of detecting patterns in programs is the vulnerability discovery method proposed by the MLSEC research group [3]. It assists the security analyst during auditing of source code, by extracting ASTs from the programs and determining structural patterns in them. Given a known vulnerability, their patterns are identified and extrapolated to a code base, such that functions potentially suffering from the same flaw can be suggested to the analyst. With that method, they managed to identify 18 previously unknown vulnerabilities in the source code of the Linux kernel.

The research context of finding code patterns using machine learning algorithms is based on 3 ideas. First,

machine learning techniques can be used to build predictive models to identify patterns in data; besides, these techniques do not require domain specific knowledge about the problem domain [4]. Second, the existing open source code repositories create new opportunities for gathering massive program repositories to be analyzed. Third, the existing big data technologies and platforms facilitate the analysis of large datasets.

When processing native code, a platform to extract binary patterns should also be able to use the debug information (if any) generated by the compiler. This information would be very valuable to extract those binary patterns, which may 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 its 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 highly parallelized and efficient tools for extracting those patterns.

A platform capable of extracting patterns from programs should also be highly parameterized. The individuals (rows in the dataset generated) to be detected by the platform must be defined 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> <generic ax>,<any>` to represent the occurrence of any 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 or row), then the corresponding value in the dataset (row and column) will be 1.

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 [5]. 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 code outside their basic blocks, representing subgraph structures (Section 3.7). 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 [6]. This kind of binary code analysis has been used for different purposes such as function entry points detection [6], compiler recognition [7], authorship attribution [8], and malware detection [9, 10]. 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.

The main contribution of this paper is a platform for the automatic extraction of patterns in native code. 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. We present an evaluation of binary pattern extraction, measuring the execution time of different configurations for a large number of programs. The parallelization provides significant performance improvements, and its efficiency is maintained for big volumes of programs.

The rest of this paper is structured as follows. Section 2 describes a motivating example, and the platform is described in Section 3. Section 4 presents an evaluation of the platform and Section 5 discusses the related work. The conclusions and future work are presented in Section 6.

2. Motivating Example

We use a motivating example to explain our platform. The example is the extraction of patterns in native code that can be later used to improve the information inferred by a decompiler. A decompiler extracts high-level information from a native program, aimed at obtaining the original source program used to generate the native code. Existing decompilers are able to infer part of this information. However, some elements of the original high-level source programs are not inferred by any decompiler.

Algorithm 1 shows a C function that returns a string (`char*` in C). The function returns a substring from the `str` parameter, starting in the `begin`th position of `str` up to the `end`th position. The values of `begin` and `end` could be negative, following the slicing behavior of the Python [] operator.

After compiling the `str_slice` function with Microsoft's `cl` compiler, the decompiled function generated by Hex-Ray 1.5 has the following signature:

```
int __cdec1 sub_401780 (int al, int q2, int a3)
```

We can see how the original return type of the function (`char*`) is not the same as the one obtained by the decompiler (`int`). This is because, in native machine code, there is no type difference between integers and pointers. The difference between both might be obtained by analyzing how the programmer uses the value returned by the function. In general, the programmer performs indirections with pointers, but not with integers. Since this rule is not always fulfilled, the decompiler does not tell the difference between these two types.

By analyzing the usage patterns of each variable, a decompiler may infer the actual high-level type of the variables (and functions). Since this is not a deterministic mechanism, the use of machine learning seems to be appropriate for this kind of problems [11]. Some recognized limitations of most decompilers include the following:

(i) Types of variables, including function arguments and return types (e.g., the example in Algorithm 1) [12].

(ii) Functions: the identification of function entry points is a complex task, mainly due to indirect function

```
char * str_slice(const char * str, int begin, int end) {
    /* Check str */
    if (str == NULL)
        return NULL;
    /* Check ranges */
    int s = strlen(str);
    if ((!s) || (begin >= s) || (begin < s * -1) || (end >= s + 1)
            || (end < s * -1))
        return NULL;
    /* Normalice values */
    size_t n_begin = begin >= 0 ? begin: s + begin;
    size_t n_end = end >= 0 ? end: s + end;
    /* Check begin >= end */
    if (n_end <= n_begin)
        return NULL;
    /* Alloc mem */
    size_t amount = n_end - n_begin;
    char * new_str = malloc((amount + 1) * sizeof(char));
    if (new_str == NULL)
        return NULL;
    /* Copy */
    memcpy(new_str, str + n_begin, amount);
    new_str[amount] = 0;
    return new_str;
}
```

Algorithm 1: Example high-level C program.

invocations [6]. Similarly, detecting the function body is an open challenge because its instructions may not be contiguous, have multiple entry points, be in-line, or not be reachable [5].

(iii) Control flow structures: its recognition is commonly based on control flow graph (CFG) analysis [12]. However, CFGs might not be completely recovered by static analysis if indirect jumps appear, which are typically generated for switch structures [13].

(iv) Elements of a specific paradigm: when another paradigm different to the structured one (e.g., object-orientation or functional) is used, the specific elements of that paradigm are barely recognized. For example, C++ decompilers commonly fail in the reconstruction of polymorphic classes, class hierarchies, member functions, and exception handling constructs [14].

The platform presented in this paper is being used to extract binary patterns from native code, which are later used to improve certain high-level information gathered by existing decompilers. Particularly, we face the problem of detecting the type returned by a function. An excerpt of the dataset generated by our platform for this particular case is shown in Table 2: individuals (rows) are functions in the module; features (columns) are sequences of binary patterns found at the end of the function body (*return* patterns) and after invoking the function (*call post* patterns); the target column is the returned type. Please, notice that the work of this paper is the platform itself, not in the algorithm for decompilation improvement.

3. Platform Architecture

This platform generates one dataset table to classify fragments of binary code (individuals or rows in the dataset) by considering the occurrence of a finite set of binary patterns (features or columns in the dataset). All the individuals and patterns (features) are obtained from a collection of binary programs, which are processed by the platform.

In our motivating example, the individuals in the dataset are functions; and the features are the generalized assembly code patterns extracted by the platform. The classification variable (the target) is the return type (we consider all the C built-in types; for compound types (structs, unions, pointers, and arrays), we only consider the type constructor, e.g., `int*` and `char**` are classified as *pointers*) of each function (individual). The generated dataset may be used later to build a machine learning model that classifies the return type depending on the patterns found in the binary code.

The platform has two working modes. The most versatile is the one shown in Figure 1. The system receives the high-level source program that will be used to generate the binary application. In this mode, the platform allows instrumenting the high-level program and 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 3.6.

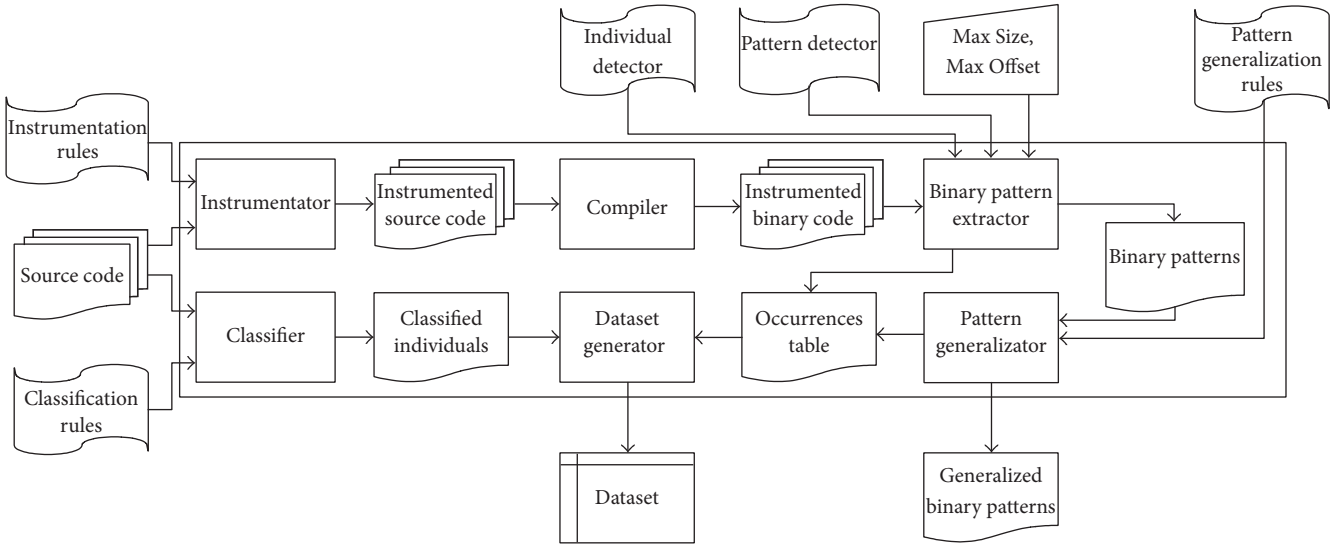

Figure 1: Platform architecture, receiving high-level code.

```
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
        end if
      end for
    end if
  end for
end
```

Algorithm 2: Instrumentation rule for return statements.

3.1. 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 3.2), 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. 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, which will be the new input for the compiler (next module in Figure 1).

In our motivating example, we have defined the instrumentation rule shown in the pseudocode in Algorithm 2 (in Section 4.1 we describe how they are implemented). For all the `return` statements in a program, the rule adds a dummy label before the statement. This label has the function identifier (id_{func}) followed by a consecutive number (a function body may have different `return` statements). This label will be searched later in the binary code (using the debug information) to know the binary instructions generated by the compiler for the `return` high-level statements. These binary instructions will be used to identify the binary patterns (Section 3.2).

Algorithm 3 shows another example of one instrumentation rule. Recall that the previous instrumentation rule was aimed at finding binary instructions between a `__RETURN_` label and a `RET` assembly instruction. However, C functions

```
Function procedure_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
    end if
  end for
end
```

Algorithm 3: Instrumentation rule for procedures.

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

Algorithm 4: Individual detector to recognize functions.

returning `void` usually do not have an ending `return` statement. Therefore, the instrumentation rule in Algorithm 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 3.2). Another typical approach is adding innocuous sequences of assembly instructions (e.g., `NOP`s) 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 to the patterns he/she wants to extract.

With the *Instrumentation Rules*, the *source code* is translated into *instrumented code*. The *instrumented code* is then compiled, producing the *Instrumented Binary Code*.

3.2. Binary Pattern Extractor. This module performs 3 tasks. First, it identifies the binary code fragments representing the individuals (rows) in the generated dataset. Second, it extracts the binary patterns (columns) detected for each individual. These 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 3 tasks.

The *Individual Detector* initially recognizes each individual in the binary code. It must implement a function to collect all the individuals. Algorithm 4 shows the *Individual Detector* of our example, recognizing functions as individuals. In the figure, `is_function` returns whether the parameter is the first instruction in a function, using the debug information generated by the compiler. Once one function is detected, its label is added to the `individuals` list, the returned value.

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

Algorithm 5 presents a *Pattern Detector* of 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`.

Algorithm 6 shows another Pattern Detector used in our example. It detects as a pattern the instructions after one `CALL` (we call it *call post*). In this case, the individual associated with the pattern is not the function the instruction belongs to, but the function being called. Similarly, we have also specified a pattern with the instructions before `CALL`, called *call pre*, not shown in the algorithm. The idea of these two patterns

```
Function RET_pattern_detector(instruction)
  if instruction is not __RETURN_id_func_n then
      return null
  end if
  begin_instruction ← instruction
  while not instruction is RET do
      instruction ← next(instruction)
  end while
  return (id_func, (begin_instruction, next(instruction)))
end
```

Algorithm 5: Pattern detector rule to recognize RET patterns.

```
Function CALL_Post_pattern_detector(instruction)
  if instruction is CALL id_func then
      begin_instruction ← instruction
      for i = 0 to MaxSize + MaxOffset do
          instruction ← next(instruction)
      end for
      return(id_func, (begin_instruction, instruction))
  end if
  return null
end
```

Algorithm 6: Pattern detector rule to recognize *call post* patterns.

is that the usage of the value returned by a function (*call post*) and the code to push its parameters (*call pre*) may be valuable to infer the types of 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 *call pre* and *call post* 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).

The algorithm to obtain the subpatterns is parameterized by the *Max Size* and *Max Offset* parameters shown in Figure 1. This algorithm starts with one-instruction length subpatterns (*size* = 1), increasing this value up to *Max Size* 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 *Max Offset*.

Figure 2 shows 4 example subpatterns. From a *call pre* pattern the *size* = 5 and *offset* = 0 and *size* = 2 and *offset* = −2 are shown. From another *call post* pattern, Figure 2 displays the *size* = 3 and *offset* = 2 and *size* = 3 and *offset* = 1 subpatterns.

The last task to be undertaken by the *Pattern Detector* is to associate the individuals with their patterns and make this association explicit by writing the *Occurrences Table*. This process is done generating as many table rows as individuals found by the *Individual Detector* (in Algorithm 4), and associating them with the rows representing each of the subpatterns found for that individual by the *Pattern Detector* functions (Algorithms 5 and 6).

3.3. Pattern Generalizator. Sometimes, the subpatterns found are too specific. For example, the `MOV eax,5` and `MOV ax,1` subpatterns are recognized as two different ones. However, for detecting whether a function is returning a value or not, they may be considered as the same pattern, meaning that a literal has been assigned to the accumulator register (i.e., a `MOV <generic ax>,<literal>` pattern). To this end, the objective of the *Pattern Generalizator* module is to allow the user to reduce the number of subpatterns, by generalizing them.

This necessity of generalizing (or normalizing) assembly instructions for binary pattern extraction was already detected in previous works. In [6], the `*` wildcard matches any one instruction, and the absence of an operand means any value. In further works, they also identify the necessity of eliding memory addresses and literal values [7]. The generalization proposed by Bao et al. uses regular expressions to generalize literal values and even instruction mnemonics [5].

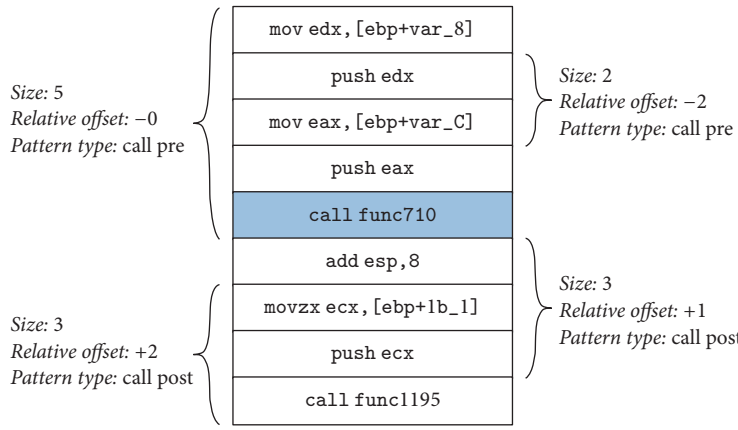

Figure 2: Example of 4 subpatterns extracted from 2 patterns.

Table 1: Example generalization of subpatterns.

	Example pattern		Generalization
Operand	`mov 5,eax`	→	`mov <literal>,<generic ax>`
	`mov [ecx],al`	→	`mov [ecx],<register>`
	`movsd xmm0,var_0`	→	`movsd xmm0,<var>`
	`mov edx,[ebp+var_1]`	→	`mov edx,[<var>]`
	`call func1493`	→	`call <address>`
Mnemonic	`movzx eax,al`	→	`<mov> <generic ax>,<any>`
	`movss [esp+54h+var_2],xmm0`	→	`<mov> [esp+54h+var_2],xmm0`
	`movsd xmm0,var_3`	→	`<mov> xmm0,<var>`
	`mov edx,[ebp+var_4]`	→	`<mov> edx,[<var>]`
	`movsx ecx,[ebp+var_5]`	→	`<mov> ecx,[<var>]`
Instruction group	`pop esi; mov esp,ebp; pop ebp; retn`	→	`<callee epilogue>`
	`mov esp,ebp; pop ebp; retn`	→	`<callee epilogue>`
	`pop ebp; retn`	→	`<callee epilogue>`
	`call func123; add esp,8`	→	`<caller epilogue>`
	`call func123`	→	`<caller epilogue>`

Table 2: Example dataset generated to predict the returned type of functions.

	(Return pattern) `<mov> <generic ax>,<any>; <callee epilogue>;`	*(Call post pattern)* `<caller epilogue>; Fld <any>;`	⋯	Return type (target)
func_710	1	0		int
func_1195	0	1		double
func_295	0	0		long long
⋮				

Another coarser normalization just ignores all the operands of assembly instructions [15].

To identify the generalization requirements of a generic platform, we analyzed the decompiler case scenario described in Section 2. Some examples of those generalizations are shown in Table 1. First, the user should be able to generalize instruction operands, including literals, registers, variables, and memory addresses. Second, the platform should allow the generalization of instructions with the same purpose. Finally, the user may need to generalize variable-instruction-length subpatterns, such as function caller and callee epilogues.

The analysis of the decompiler use case indicates that a highly expressive generalization mechanism should be

```
Function MOVE_generalization(instruction)
  if instruction.mnemonic in [mov, movzx, movsd, movss, movsx]
        and instruction.operands[1].type = register
        and instruction.operands[1].value in [eax, ax, ah, al] then
    instruction.mnemonic ← <mov>
    instruction.operands[1].value ← <generic ax>
    instruction.operands[2].value ← <any>
  end if
  return (instruction, next(instruction))
end
```

Algorithm 7: 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]
    end if
  end for
  return individuals
end
```

Algorithm 8: Classification rule associating each function with its return type.

provided by a generic binary pattern extraction platform. For instance, it should allow the generalization of variable-length groups of instructions, not supported by the existing approaches. Therefore, we propose a programmatic system that takes advantage of the expressiveness of a full-fledged programming language to describe those generalizations.

In our platform, generalizations are expressed as *Pattern Generalization Rules*. As shown in Algorithm 7, those rules are implemented as functions receiving one instruction and returning their generalized pattern (the current instruction if no generalization is required) and the following instruction to be analyzed. This second value allows the implementation of variable-instruction-length generalizations. The rule in Algorithm 7 generalizes the *move* instructions that save into the accumulator register any value.

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

3.4. 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 value of the classifier variable. These associations are described by the user with the *Classification Rules*.

Algorithm 8 shows one *Classification Rule* for our example. We iterate along the statements in the program. For each function, we associate its identifier with the returned type, which is the classifier variable for our problem (we predict the return type of functions).

3.5. Dataset Generator. Finally, the *Dataset Generator* generates the dataset from the *Occurrence Table* (Section 3.3) and the individual classification (Section 3.4): 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 in the individual. Classifier or target cells may have different values. Table 2 shows an example dataset.

3.6. Processing Binary Files. 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. [6] as an example. They extract patterns from stripped binary files to detect function entry points (FEP), which existing dissemblers do not detect perfectly yet [5]. They 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) [16]. 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 MSVS compilers [6].

Figure 3 shows the changes to the platform architecture when we want to process binary files, and the high-level program is not available. White elements are the same as in

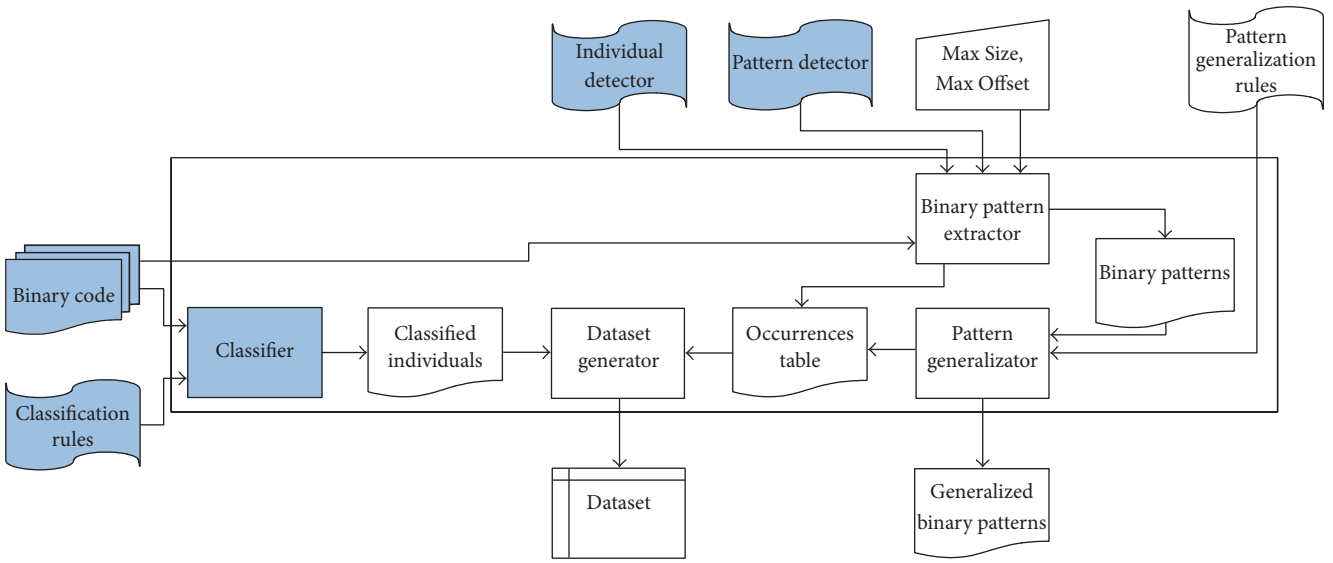

Figure 3: Platform architecture to process binary code.

the previous architecture. Blue elements are 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 different. 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 valid 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. In this second case, the feature is not associated with the offset where the pattern is detected, but to the offset (memory address) being called (as done in Algorithm 6). *Pattern Generalization* is done as the normalization process described in [6]. Finally, the *Classification Rules* use the debug information to set 1 to one individual identified as a FEP and 0 otherwise.

This platform configuration (and the previous one) to extract datasets valid to create the CRF model proposed by Rosenblum et al. is available for download at [17].

3.7. Representing Nonsequential Patterns. In the analysis of binary applications, it is common to require the detection of nonsequential 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 1 and 3) is aimed at extracting patterns made up of contiguous binary instructions, the rest of the modules can be used to represent nonsequential 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. Algorithm 5 shows how `RET` features are associated with the function (individual) where the pattern was detected. In Algorithm 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, `CALL` patterns are associated with individuals in a different way. Algorithm 6 shows how this type of feature is not associated with the function where the pattern is detected, but to the function being invoked. Therefore, a machine learning algorithm trained with the generated dataset may associate nonsequential patterns (e.g., there must exist a `RET` pattern inside the function *and*, in any part of the program, a `CALL` pattern invoking the same function) to identify the type returned by a function.

Another example of this functionality is the FEP identification problem described in Section 3.6. The dataset generated by out platform can be used to create the proposed CRF model, which uses graphs for structural prediction and

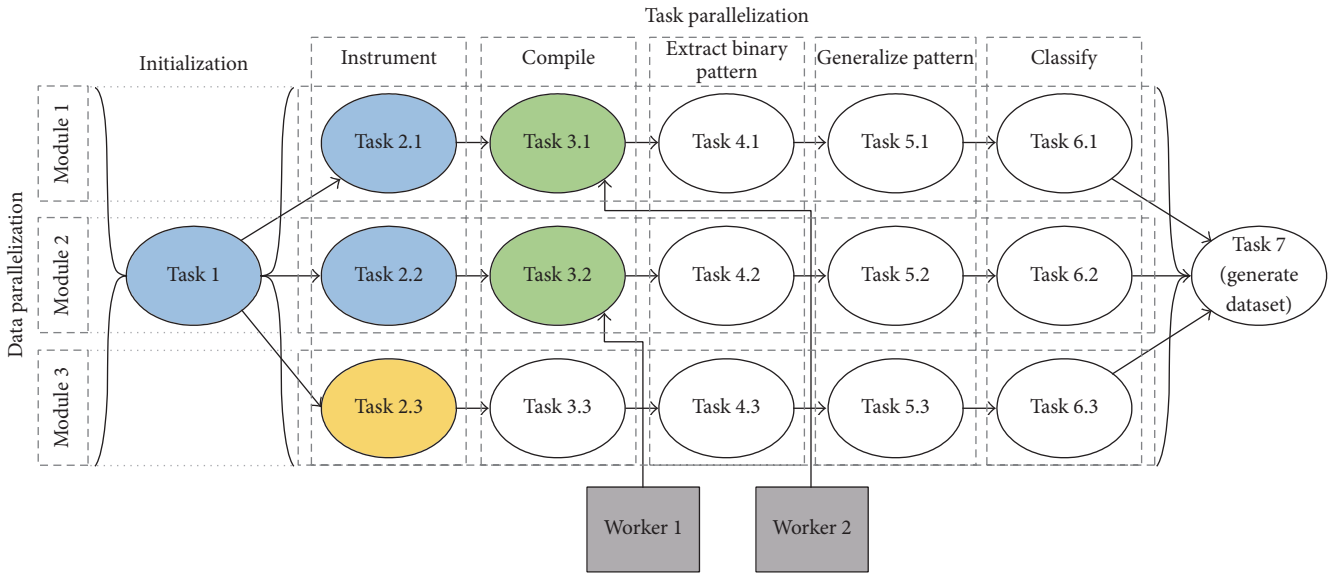

Figure 4: Parallelization of the platform implementation.

classification [16]. Those graphs are obtained from the dataset by using the versatile association of features to individuals already discussed. Its implementation and a sample dataset can be consulted in [17].

4. Evaluation

4.1. Platform Implementation. We have implemented the proposed platform and it is freely available at http://www.reflection.uniovi.es/decompilation/download/2016/sp/. The *Instrumentator* and *Classifier* modules have been implemented in C++, since they use *clang* [18] to process the high-level representation of C programs. The rest of the platform has been implemented in Python. For the disassembly services we have used IDAPython [19].

The implementation is highly parallelized, providing important performance benefits when multicore architectures are used. The parallelization follows a pipeline scheme, where both data and task parallelism are used. Figure 4 shows the concrete approach followed. These have been the issues 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. This `obj` files can be combined in `lib` or `exe` files to produce bigger modules. In the example in Figure 4, 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). As shown in Figure 4, 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 defined 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 4, 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 complete. 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 [20], we implemented tasks as processes to obtain a better runtime performance improvement with multicore architectures [21].

(5) Concurrent Workers. To parameterize the level of parallelization of the platform, we configured its implementation to run with a different number of worker processes (Section 4.2). A scheduler analyzes the task DAG and tells each worker which is the following task to be executed. In Figure 4, two workers are running in parallel. Tasks 1, 2.1, and 2.2 have already been executed; Tasks 3.1 and 3.2 are run by Workers 1 and 2, respectively; and Task 2.3 is the following one to be executed, once one worker is free.

(6) Communication between Tasks. Since we implemented tasks as processes, communication between them is costly. However, the dependency between tasks shown in Figure 4 indicates that the output of one task is taken as the input of the following one. Therefore, this data communication was

implemented 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 in the `multiprocessing` module.

(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 write information in the standard output (e.g., the C compiler). We captured those messages and sent them to a concurrent logger, adding additional information of the processes.

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

(i) Number or independent modules of compilation (or programs). We may process different programs in parallel, or different modules of the same program, to create a dataset.

(ii) 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.

(iii) The number of cores: we have run our platform with different multicore computers.

(iv) The size of each program (or module), according to the number of individuals it may contain.

(v) Subpattern extraction: as described in Section 3.2, different subpatterns are automatically extracted from the patterns found. The *Max Size* and *Max Offset* parameters have influence on the execution time.

(vi) The number of patterns: the proposed 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 parallelization level. In order to evaluate that, we fix all the variables except one and measure the runtime performance for different values of the free variable [22]. This process is repeated for all the variables.

We evaluate the platform with the real example of predicting the return type in binary programs, using their C source code (the first working scheme of our system, shown in Figure 1). We extract *return*, *call pre,* and *call post* patterns, divide them into different subpatterns, and perform a generalization of the subpatterns found.

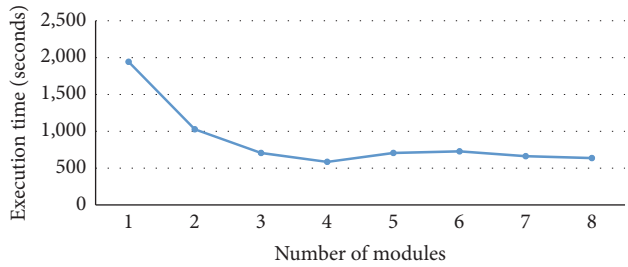

Figure 5: Execution time for an increasing number of modules.

The programs used for the experiments were synthetically generated by a C program generator. It was very helpful to generate a rich battery of programs. Besides, we were able to generate different configurations of the same program, changing the number of individuals (functions in our example) per module. In this way, we do not introduce the bias of measuring different programs.

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 8 GB 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 [23].

4.3. Increasing Number of Modules. In this first experiment, we increase the modules in a program from 1 to 8, fixing the number of cores and workers to 4. For this experiment and the following ones, the value of *Max Size* is 4 and *Max Offset* is 0. We also extract *return*, *call pre,* and *call post* 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 processed program is the same.

Figure 5 shows the benefits of parallelization. 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). In 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 for that configuration is 4 factors [24].

For more than 4 modules, there is no significant benefit, since there are only 4 cores in this configuration. Besides, there is no penalty for 8 programs, showing that a number of programs higher that the number of cores do not cause a significant penalty. 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.

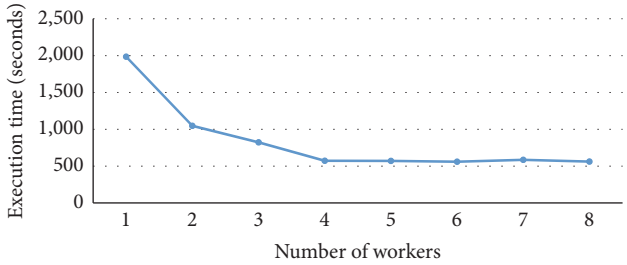

Figure 6: Execution time for an increasing number of workers.

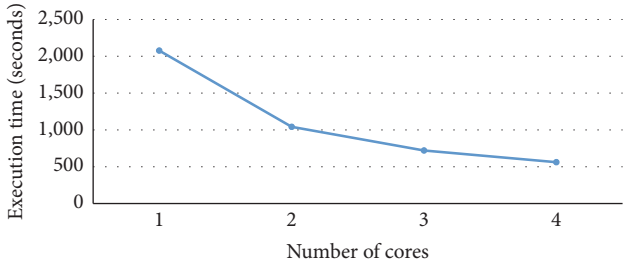

Figure 7: Execution time for an increasing number of cores.

4.4. 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 6 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.

4.5. 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 4.2), 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 7 how our platform takes advantage of multicore 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 [24].

4.6. Increasing Number of Modules and Workers. This experiment increases two variables at the same time. It is intended to represent a typical use case scenario. Assuming we have a multicore computer (4 cores in our case), we set the number of workers equal to the number of modules (or programs) to be processed. The idea is to try to obtain the higher 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, equally distributed over the different modules of the program.

Figure 8 shows how execution time keeps reducing until 4 modules and workers (3.5 factors of benefit). From 4 to 7,

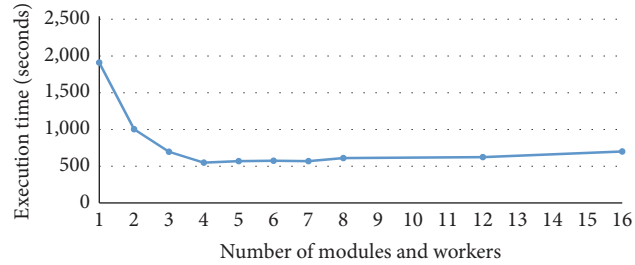

Figure 8: Execution time for an increasing number of modules and workers.

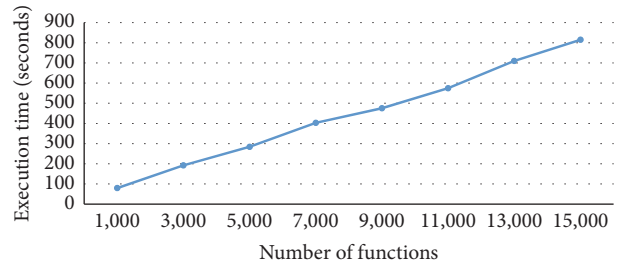

Figure 9: Execution time for an increasing number of functions.

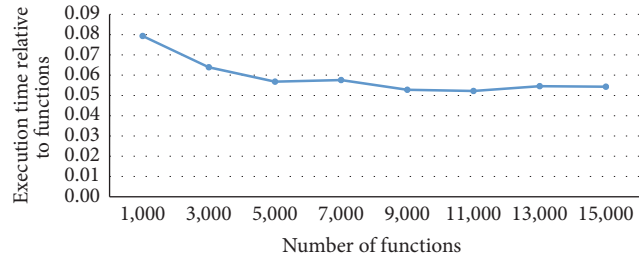

Figure 10: Execution time per function, increasing the number of functions.

differences among the values are lower than 1% (practically the same values). With 8 and beyond, the figure displays a slight increase of 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 range from the number of cores to twice this value.

4.7. Increasing Number of Functions. In order to see how the platform behaves for increasing sizes of programs, 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 4.

Figure 9 shows the linear increase of runtime performance depending on the number of functions (i.e., the size of the programs). Besides, it supports the analysis of really big modules with 15,000 functions.

Figure 10 presents another view of the same data. That figure displays the execution time performance per function, increasing the number of functions in the program. For small programs, there is an initialization penalty causing a higher

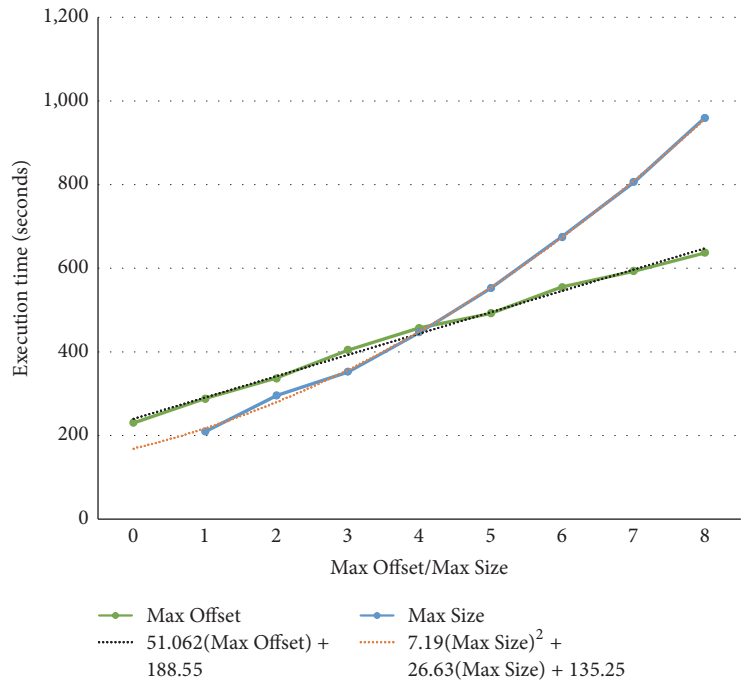

Figure 11: Execution time for an increasing number of *Max Offset* and *Max Size* parameters.

execution time to process a low number of functions. 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 performance of the platform is not decreased for big input programs.

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

Figure 11 shows both variables. We can see how *Max Offset* has a linear influence on execution time. The regression line shown in Figure 11 has a slope of 51, representing the cost in seconds of increasing one unit in *Max Offset*. For *Max Size*, the best regression obtained is quadratic (Figure 11). The user should be aware of that, meaning that choosing high values for *Max Size* may involve much greater increases of the execution times.

4.9. 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 decompiler example, we identified 3 patterns: *return*, *call pre,* and *call post*. We measure runtime performance of the 7 different combinations of these 3 patterns. Modules, workers, cores, *Max Size,* and *Max Offset* are fixed to 4, and each module contains 750 functions (3,000 in total).

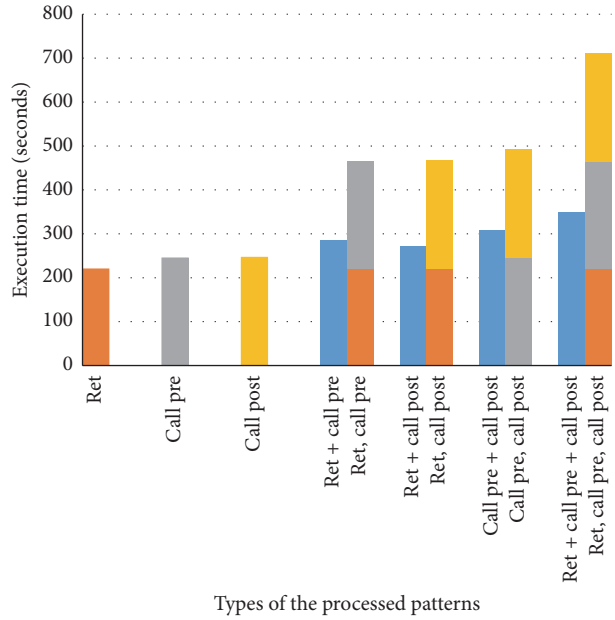

Figure 12: Execution time when extracting different types of patterns.

Figure 12 shows the results. The 3 first bars show the execution time consumed to extract each pattern individually. The 3 next bars display the execution time for two patterns in parallel, compared to the costs of extracting them individually. We can see how the platform obtains an average benefit of 1.65 factors due to the parallelization. When the platform extracts 3 patterns at the same time, this benefit increases to 2.1 factors.

4.10. 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. As mentioned, this is an existing problem of existing decompilers. The purpose of this section is not to present how this problem may be solved with machine learning, but to measure the execution time required to extract the binary patterns and to build the model.

To predict the type returned by a function, we extract binary code patterns before `ret` instructions and before and after function invocations. We found out that the number of functions required to build an accurate model for this problem is very high, so a huge program database would be needed. Instead, we implemented a code generation tool that writes synthetic C functions considering the language grammar and its type system. This way, we can generate any number of random functions (and invocations to them) for all the different types in the language (C built-in types plus type constructors for compound types (structs, unions, pointers, and arrays)). These functions are then passed to our platform to generate the output dataset. Then, the dataset is used to build a J48 classifier using Weka.

As mentioned, we can generate any number of C functions to be passed to our platform. Therefore, 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 that functions with our platform; and we use Weka with the generated dataset to compute the accuracy rate using 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 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 J48 model with the dataset generated in the last iteration.

Following the method described above, we created a dataset with 160,000 functions and 3,321 binary patterns (the dataset file 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. For comparison purposes, we also evaluated the execution time to build a J48 model with the 160,000-function dataset, taking 11 hours, 55 minutes, and 15 seconds to build the model. Notice that Weka builds the model sequentially, not taking advantage of all the cores in the system.

5. Related Work

There exist different works aimed at extracting assembly patterns from existing applications. To the knowledge of the authors, none of them have built a platform to extract those patterns automatically. They define custom processes and, some of them, even manual procedures.

Rosenblum et al. extract every combination of 1, 2, and 3 consecutive assembly instructions from a big set of executable files [6]. Then, they use forward feature selection to filter the most significant patterns and later train a Conditional Random Fields to detect the function entry points. The same authors use this methodology to detect the compiler used to generate the executables [7]. This research work was later extended to consider the compiler options and programming language used in the source application [8], to identify the programmer that coded the application [25], and to identify the functions belonging to the operating system [26].

BYTEWEIGHT provides another approach to find function entry points [5]. They apply machine learning to recognize the patterns, so that different compilers and optimization options may be used. Analyzing the training binaries, an extraction process generates prefix trees from sequences of bytes or normalized instructions. The prefix tree represents possible function start sequences. Then, they assign a weight representing the likelihood that the path from the root to the node is a function start in the training set. Finally, the weighted prefix tree is used to classify the input binary file.

Apart from assembly patterns extraction, there are situations where other parts of the binary files need to be processed. One example is the detection of packed executable files [27]. To this end, it is necessary to recognize not only assembly patterns, but also other types of information existing in the binaries, such as header patterns, entropy values, and characteristics of the file sections. Ugarte-Pedrero et al. propose a custom collective-learning-based process to solve this problem, detecting packed executables upon structural features, and heuristics [28].

Regarding decompilation, Cifuentes et al. identified the existing limitations on recognizing high-level control structures [29]. They later define a technique to recover jump tables and their target addresses and incorporated it in the DCC decompiler [30]. The Phoenix decompiler uses a structuring algorithm to detect control flow structures, being able to decompile more structures than Hex-Rays [13]. Regarding decompilation of high-level types, Mycroft proposes a constraint-based algorithm to infer types from binary code [31]. Another type recovery approach is the VSA algorithm based on value propagation [32]. Laika is a system that uses Bayesian unsupervised learning to detect high-level data structures, analyzing the process memory images [33].

6. Conclusions

We propose a platform for the automatic extraction of patterns in binary files, capable of analyzing big executable files. The platform is highly parameterized to be used in different scenarios. The extracted patterns can be used to predict features in native code, when the high-level source code and the debug information are not available.

The platform implementation has been parallelized to increase its runtime performance on multicore architectures. Both data and task parallelization schemes have been followed. We have evaluated its performance, obtaining a performance benefit of 3.5 factors over the maximum theoretical value of 4 factors. The evaluation presented also documents how the different parameters of the platform should be used to obtain the best performance.

We are currently using the proposed platform to extract patterns that are later used to improve the information inferred by existing decompilers. We generate patterns of

high-level type information to train a classifier using different machine learning algorithms. We are currently focused on the return types of functions, but we hope to apply it to parameters and local and global variables.

We plan to use clustering algorithms to the dataset generated for a big battery of programs taken from open source code repositories. The objective is to obtain classifications of (sections of) applications depending on the patterns found inside them. The classes obtained may be helpful to identify the code that performs common input/output, network, and computing intensive or multithreaded operations.

The platform implementation, its source code, the 3 different configurations used in this article (return type, function or procedure identification, and FEPs extraction in binary files), and all the examples used in the evaluation are available for download at http://www.reflection.uniovi.es/decompilation/download/2016/sp/.

Competing Interests

The authors declare that they have no competing interests.

Acknowledgments

This work 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). The authors have also received funds from the Banco Santander through its support to the Campus of International Excellence.

References

[1] Defense Advanced Research Projects Agency, MUSE envisions mining "big code" to improve software reliability and construction, 2014, http://www.darpa.mil/news-events/2014-03-06a.

[2] F. Ortin, J. Escalada, and O. Rodriguez-Prieto, "Big code: new opportunities for improving software construction," *Journal of Software*, vol. 11, no. 11, pp. 1083–1008, 2016.

[3] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*, pp. 359–368, ACM, Los Angeles, Calif, USA, December 2012.

[4] E. Alpaydin, *Introduction to Machine Learning*, The MIT Press, 2nd edition, 2010.

[5] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: learning to recognize functions in binary code," in *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC '14)*, pp. 845–860, USENIX Association, San Diego, Calif, USA, August 2014.

[6] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt, "Learning to analyze binary computer code," in *Proceedings of the 23rd National Conference on Artificial Intelligence—Volume 2 (AAAI '08)*, pp. 798–804, AAAI Press, 2008.

[7] N. E. Rosenblum, B. P. Miller, and X. 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)*, pp. 21–28, ACM, Toronto, Canada, June 2010.

[8] N. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA '11)*, pp. 100–110, ACM, Ontario, Canada, July 2011.

[9] I. Santos, Y. K. Penya, J. Devesa, and P. G. Bringas, "N-grams-based file signatures for malware detection," in *Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS '09)*, pp. 317–320, AIDSS, 2009.

[10] C. Liangboonprakong and O. Sornil, "Classification of malware families based on N-grams sequential pattern features," in *Proceedings of the 8th IEEE Conference on Industrial Electronics and Applications (ICIEA '13)*, pp. 777–782, June 2013.

[11] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from 'big code'," in *Proceedings of the 42nd Annual ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL '15)*, pp. 111–124, 2015.

[12] K. Troshina, A. Chernov, and Y. Derevenets, "C decompilation: is it possible?" in *Proceedings of the International Workshop on Program Understanding (PSI '09)*, pp. 18–27, Altai Mountains, Russia, 2009.

[13] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Proceedings of the 22nd USENIX Security Symposium, USENIX*, pp. 353–368, Washington, DC, USA, 2013.

[14] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina, "SmartDec: approaching C^{++} decompilation," in *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE '11)*, pp. 347–356, IEEE, October 2011.

[15] Y. Fan, Y. Ye, and L. Chen, "Malicious sequential pattern mining for automatic malware detection," *Expert Systems with Applications*, vol. 52, pp. 16–25, 2016.

[16] J. D. Lafferty, A. McCallum, and F. C. N. Pereira, "Conditional random fields: probabilistic models for segmenting and labeling sequence data," in *Proceedings of the 18th International Conference on Machine Learning (ICML '01)*, pp. 282–289, Morgan Kaufmann, 2001.

[17] J. Escalada and F. Ortin, Source code for the article: An efficient platform for the automatic extraction of patterns in native code, 2016, http://www.reflection.uniovi.es/decompilation/download/2016/sp.

[18] LLVM, clang: a C language family frontend for LLVM, 2016, http://clang.llvm.org.

[19] E. Bachaalany, GitHub: IDAPython, 2016, https://github.com/idapython.

[20] D. Beazley, "Understanding the python GIL," in *Proceedings of the PyCON Python Conference*, Atlanta, Ga, USA, February 2010.

[21] D. Phillips, *Python 3 Object-Oriented Programming*, Packt Publishing Ltd, Livery Place, Birmingham, UK, 2nd edition, 2015.

[22] J. M. Redondo, F. Ortin, and J. M. C. Lovelle, "Optimizing reflective primitives of dynamic languages," *International Journal of Software Engineering and Knowledge Engineering*, vol. 18, no. 6, pp. 759–783, 2008.

[23] F. Ortin, L. Vinuesa, and J. M. Felix, "The DSAW aspect-oriented software development platform," *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, no. 7, pp. 891–929, 2011.

[24] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of*

the Spring Joint Computer Conference, pp. 483–485, Atlantic City, NJ, USA, April 1967.

[25] N. Rosenblum, X. Zhu, and B. P. Miller, "Who wrote this code? Identifying the authors of program binaries," in *Computer Security—ESORICS 2011: 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12–14,2011. Proceedings*, vol. 6879 of *Lecture Notes in Computer Science*, pp. 172–189, Springer, Berlin, Germany, 2011.

[26] E. R. Jacobson, N. Rosenblum, and B. 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)*, pp. 1–8, ACM, Szeged, Hungary, September 2011.

[27] I. Santos, X. Ugarte-Pedrero, B. Sanz, C. Laorden, and P. G. Bringas, "Collective classification for packed executable identification," in *Proceedings of the 8th Annual Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference (CEAS '11)*, pp. 23–30, Perth, Australia, September 2011.

[28] X. Ugarte-Pedrero, I. Santos, and P. G. Bringas, "Structural feature based anomaly detection for packed executable identification," in *Computational Intelligence in Security for Information Systems: 4th International Conference, CISIS 2011, Held at IWANN 2011, Torremolinos-Málaga, Spain, June 8–10, 2011. Proceedings*, vol. 6694 of *Lecture Notes in Computer Science*, pp. 230–237, Springer, Berlin, Germany, 2011.

[29] C. Cifuentes, D. Simon, and A. Fraboulet, "Assembly to high-level language translation," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '98)*, pp. 228–237, IEEE, Bethesda, Md, USA, November 1998.

[30] C. Cifuentes and M. Van Emmerik, "Recovery of jump table case statements from binary code," *Science of Computer Programming*, vol. 40, no. 2-3, pp. 171–188, 2001.

[31] A. Mycroft, "Type-based decompilation," in *Proceedings of the European Symposium on Programming (ESOP '99)*, pp. 208–223, 1999.

[32] G. Balakrishnan and T. Reps, "Divine: discovering variables in executables," in *Verification, Model Checking, and Abstract Interpretation: 8th International Conference, VMCAI 2007, Nice, France, January 14–16, 2007. Proceedings*, vol. 4349 of *Lecture Notes in Computer Science*, pp. 1–28, Springer, Berlin, Germany, 2007.

[33] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures," in *Proceedings of the 8th Conference on Operating Systems Design and Implementation (OSDI '08)*, pp. 255–266, San Diego, Calif, USA, December 2008.