Master's Thesis

Automatic stub generation from natural language description

1410212 Le Vinh

Supervisor: Prof. Mizuhito Ogawa
Main Examiner: Prof. Mizuhito Ogawa
Examiners: Associate Prof. Nao Hirokawa
Associate Prof. Nguyen Minh Le

School of Information Science
Japan Advanced Institute of Science and Technology

August 2016

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Mizuhito Ogawa for the continuous support of not only my thesis but also my life. He instructed me how to deal with a problem, and evaluate research.

My sincere thanks also goes to Associate Professor Nao Hirokawa and Associate Professor Nguyen Minh Le for their detailed and constructive comments and also for their important support through this work. I learned a lot from him how to write scientific documents.

I would like to thank to Mr Nguyen Minh Hai and my laboratory colleagues, who helped me a lot in the research. I owe my loving thanks to my parents and my friends. Without their encouragement and understanding it would have been impossible for me to finish this work.

Contents

Chapter 1

Introduction

Motivation and problems

Malware is a computer program which is intended to damage or disrupt a system, some typical types of malware are virus, trojan horse and keylogger [1]. It is distributed as a binary executable, without source codes. The advanced techniques, such as obfuscation techniques (e.g. dead code insertion, code reordering, instruction replacement), polymorphic techniques (e.g. self encryption and self modification) and simulator detection (e.g. change behavior in emulation environment), have been used in malwares, which make them more difficult to be detected by signature recognition and virtual emulation methods.

Model checking based approaches for malware dection attract many attention. The binary executable is analyzed to infer an abstract model, such as Control Flow Graphs (CFG), and then analysis techniques based on model checking can be adopted [2, 3, 4, 5]. However, constructing CFG for a malware binary program is also a challenge due to the confusion of obfuscation techniques.

There are various model generation tools from binary executables, such as BIRD [6], CodeSurfer/x86 [7], BINCOA/OSMOSE [8, 9], Renovo [10], and Syman [11]. However, only Syman supports system calls with a Window API emulator.

BE-PUM (Binary Emulation for PUshdown Model generation) is a binary analyzer [12]. Currently, BE-PUM focuses on malware programs. BE-PUM applies symbolic execution to execute a program, and output Control Flow Graph for input binary executable file. API stub is used to handle system API calls. Each Windows API call is treated as a single instruction, and after that the environments is updated as the technical document from Microsoft Developer Network. However, by observation, API stub does not affect the path condition. API stub is a proxy object that can invoke API native function and update the simulation environment after API call. It makes flexibility in symbolic execution, and avoids the cost of manual APIs approximation.

There remain several problems need to be solved. For instance, loop handling in efficiency, huge engineering effort for implementation in symbolic execution. We are taking on the last problem, where there are about 1000 x86 instruction and more than 4000 Windows API. BE-PUM implementation requires binary emulation for the former and

API stub for the latter. Currently, there are about 400 APIs implemented manually, and the lack of APIs stub may lead the unexpected termination of BE-PUM. To reduce manual effort and increase the ability of BE-PUM, our target is to automatically generate Windows API Stub from Windows APIs description in natural language.

Contribution

We observes that Windows API stub generation from natural language description requires limited API specifications. Besides, testing with executable environments can be applied to avoid ambiguity in specification extracted from natural language description. Currently, we have constructed a system for API stub generation.

- The system can automatically collect API and structure descriptions from Microsoft Developer Network (MSDN). Currently, we have collected about 1800 API description.
- The system can extract API specification and generate API stub. There are about 1200 API stubs that were generated. It is more than three times the number of manual API stubs in current BE-PUM.
- The generated API stubs allow BE-PUM to interpret more system APIs call, and analysis malwares that are unsupported by the current BE-PUM.

Thesis Outline

The thesis is organized in 8 chapters: Introduction, Preliminaries, BE-PUM System, Observation, Specification Extraction, Stub Generation, Implementation and Conclusion. Chapter 2 introduces the background knowledge related to the research. Chapter 3 briefly presents BE-PUM system. Chapter 4 expresses important observations to decide our implementation choice. Chapter 5 and Chapter 6 describe our methodology to extract API specification and generate API stub. Chapter 7 presents our implementation and experiments. Finally, Chapter 8 describes current limitations and future works.

Chapter 2

Preliminaries

2.1 Windows API

Application Programming Interface (API) is a set of instructions, functions, objects and protocols, which allow a user to build software applications or interact with an external system [1].

Windows application programming interface (Windows API) is a set of system APIs in the Microsoft Windows operating systems [2] . The name Windows API covers its root in 16-bit Windows and extension to 64-bit Windows [3]. However, in this thesis we focus on x86 binary executable files over 32-bit Windows operating system.

A dynamic-link libraries (DLL) is a library containing code and data that can be used by more than one program at the same time, and the link between the program and library are linked at running time [4]. Windows API are distributed in DLLs (without source code) that are part of the Windows operating system [5]. In the research, we collect about 1800 APIs in 25 DLLs, and focus on kernel32.dll (containing about 700 APIs), gdi32.dll (containing about 270 APIs), user32.dll (containing about 170 APIs) and advapi32.dll (containing about 160 APIs).

Windows API descriptions are the documents, which contain the definitions and explanation of functions, variable types or data structures for software developers. They are published online by Microsoft Developer Network (MSDN) [6]. The description is in HTML document, and often follows a similar style: the purpose of API, the API function prototype (in C programming language), the explanation of each parameter and the return

[1] API Definition
`http://techterms.com/definition/api`

[2] Windows API
`https://en.wikipedia.org/wiki/Windows_API`

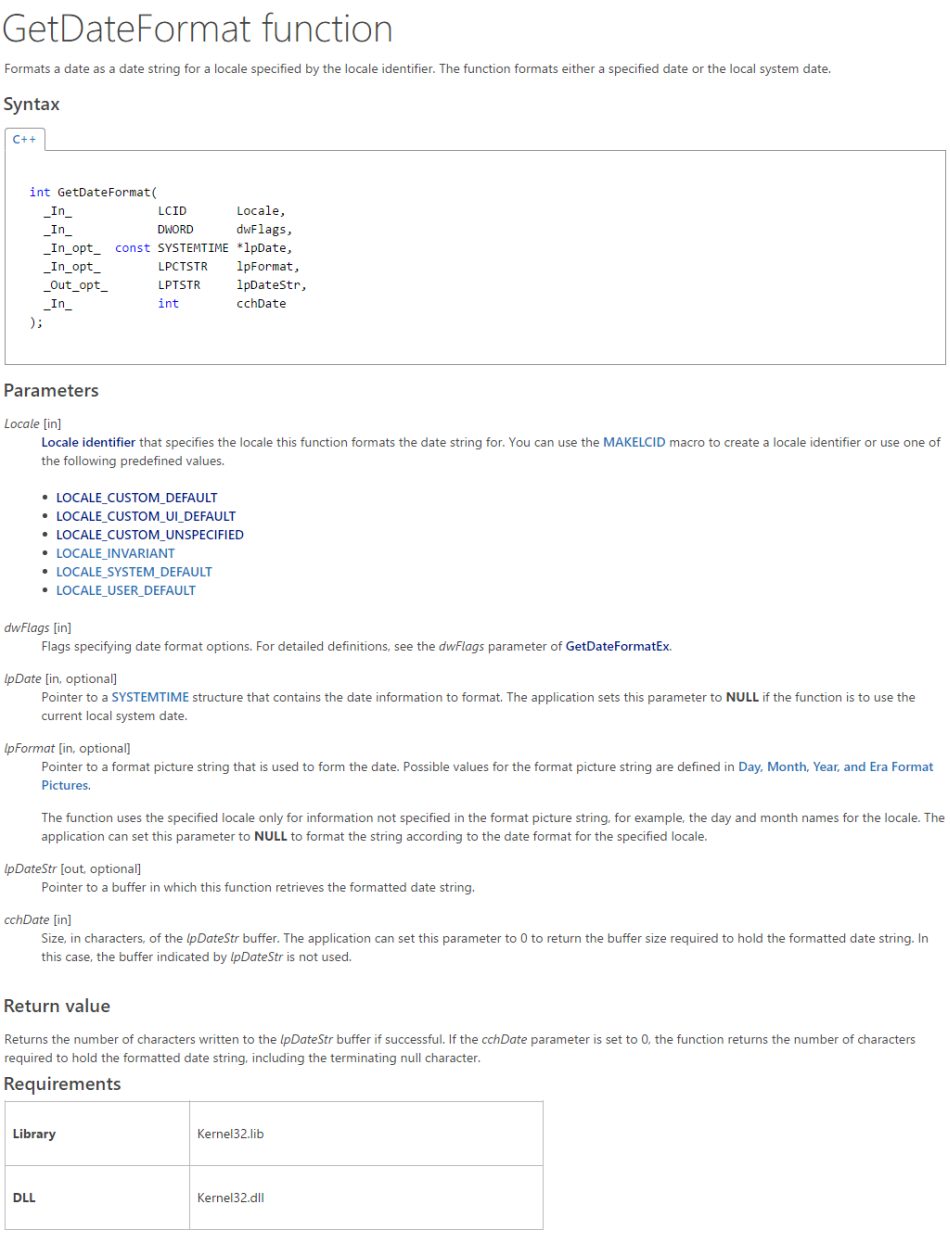
[3] Windows API Index
https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx

[4] What is a DLL?
`https://support.microsoft.com/en-us/kb/815065`

[5] Walkthrough: Calling Windows APIs
`https://msdn.microsoft.com/en-us/library/172wfck9.aspx`

[6] `https://msdn.microsoft.com/en-us`

value, the connection to related APIs or data structure description, and other information (e.g., version and DLL). For example, figure 2.1 shows the description of GetDateFormat function [7]:



GetDateFormat function

Formats a date as a date string for a locale specified by the locale identifier. The function formats either a specified date or the local system date.

Syntax

C++

```
int GetDateFormat(
  _In_            LCID       Locale,
  _In_            DWORD      dwFlags,
  _In_opt_  const SYSTEMTIME *lpDate,
  _In_opt_        LPCTSTR    lpFormat,
  _Out_opt_       LPTSTR     lpDateStr,
  _In_            int        cchDate
);
```

Parameters

Locale [in]

Locale identifier that specifies the locale this function formats the date string for. You can use the MAKELCID macro to create a locale identifier or use one of the following predefined values.

- LOCALE_CUSTOM_DEFAULT
- LOCALE_CUSTOM_UI_DEFAULT
- LOCALE_CUSTOM_UNSPECIFIED
- LOCALE_INVARIANT
- LOCALE_SYSTEM_DEFAULT
- LOCALE_USER_DEFAULT

dwFlags [in]

Flags specifying date format options. For detailed definitions, see the *dwFlags* parameter of **GetDateFormatEx**.

lpDate [in, optional]

Pointer to a SYSTEMTIME structure that contains the date information to format. The application sets this parameter to **NULL** if the function is to use the current local system date.

lpFormat [in, optional]

Pointer to a format picture string that is used to form the date. Possible values for the format picture string are defined in Day, Month, Year, and Era Format Pictures.

The function uses the specified locale only for information not specified in the format picture string, for example, the day and month names for the locale. The application can set this parameter to **NULL** to format the string according to the date format for the specified locale.

lpDateStr [out, optional]

Pointer to a buffer in which this function retrieves the formatted date string.

cchDate [in]

Size, in characters, of the *lpDateStr* buffer. The application can set this parameter to 0 to return the buffer size required to hold the formatted date string. In this case, the buffer indicated by *lpDateStr* is not used.

Return value

Returns the number of characters written to the *lpDateStr* buffer if successful. If the *cchDate* parameter is set to 0, the function returns the number of characters required to hold the formatted date string, including the terminating null character.

Requirements

Library	Kernel32.lib
DLL	Kernel32.dll

Figure 2.1: GetDateFormat description

[7] `https://msdn.microsoft.com/en-us/library/windows/desktop/dd318086(v=vs.85).aspx`

2.2 Java Native Access (JNA)

BE-PUM replies on Java Native Access (JNA) [8] library to invoke Windows APIs.

JNA is a community-developed library that provides Java programs easy access to native shared libraries containing native codes compiled for a specific system (e.g., Windows x86), and JNA allows native functions to be called directly by using natural Java method invocation [8].

In Be-PUM project, the JNA is choosen to invoke Windows API due to several reasons. Firstly, JNA does not require any additional non-Java or native code. This simplicity is an advantage, compared to Java Native Interface (JNI) [9]. Secondly, there is a package, which provides further supports for Window platform and calling Windows APIs. Furthermore, JNA is supported, maintained and recommended by a large Java developer community.

An example of Windows API call is used to present how to invoke a Windows API function via JNA. The code below shows how to call GetDateFormat function..

```
// Java code
public interface Kernel32DLL extends StdCallLibrary {
    // Library mapping
    Kernel32DLL INSTANCE = (Kernel32DLL) Native.loadLibrary("kernel32",
        Kernel32DLL.class);
    // Function mapping
    public int GetDateFormat(int Locale, int i, SYSTEMTIME lpDate,
        String lpFormat, char[] lpDateStr, int cchDate );
}
public static void main(String[] args){
    Kernel32DLL kernel32 = Kernel32DLL.INSTANCE;
    char date[] = new char[100];
    WString dFormat = new WString("dd : MMMM : yyyy");
    int ret = kernel32.GetDateFormat(2048, 0, null, dFormat, date, 100);
    // Output: ret = 17, and date = "31 : July : 2016"
}
```

Library Mapping is the step to declare the DLL containing a target API, and a class corresponding to this library need to be created. When a native library interface is instantiated by method Native.loadLibrary(), JNA creates a proxy object for this library. The proxy provides a mechanism to looks up and invoke the appropriate function object, which represents the corresponding function exported by the native library [10] .
In the above example, the proxy object for Windows Kernel32 DLL is created.

[8] JNA HomePage
https://github.com/java-native-access/jna

[9] Java programming with JNI
http://www.ibm.com/developerworks/java/tutorials/j-jni/j-jni.html

[10] Functional Overview
https://github.com/java-native-access/jna/blob/master/www/FunctionalDescription.md

Function Mapping is the step to declare methods corresponding to native functions in DLL. The method signature needs to be defined in Java interface, and it figures out function name, input parameters and return type. Native libraries, such as DLLs, often contains a lot of API functions, but only API functions actually used in the program need to be declared. JNA handles the run-time mapping of the method to the DLL function transparently, and the function is mapped directly from its method interface name to the the native library. Therefore the match between method name and function name must be ensured exactly.
Figure 2.2 shows the process of invoking GetDateFormat function.

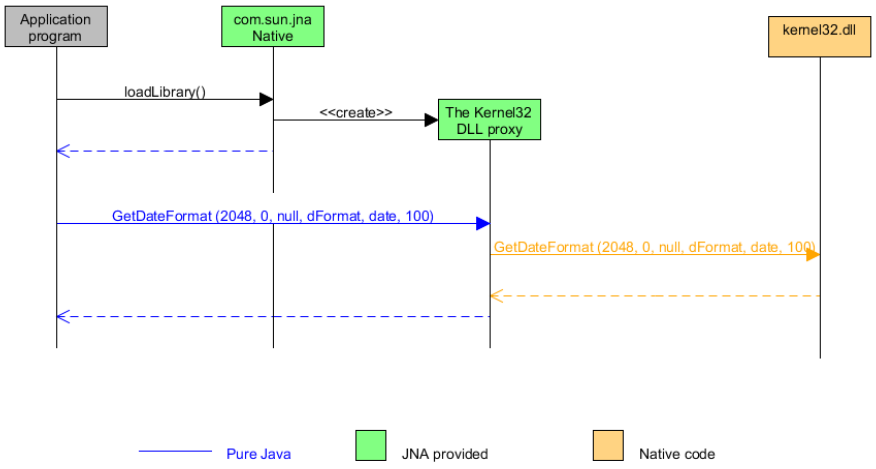


Figure 2.2: Sequence diagram for invoking GetDateFormat call

Passing parameters to native stack is taken by JNA. *In Java Virtual Machine (JVM)*, the memory model includes stacks (thread stacks and native stacks) and the heap memory area. The thread stack stores local primitives variables and references. The native stack is used to invoke native methods. The heap memory area contains objects created in the program. JNA takes care the process: copying the value of parameters from Java stack to the native stack, invoking the API function, coming back Java stack and the next instruction in Java code. The results of native API function call are the return value stored in EAX register, and the changes occurred in the heap memory.

From the perspective of binary level, there are two kinds of parameters: primitive (the value of parameter is stored in stack) and pointer (the value of parameter is a memory area pointed by a pointer).

Principle. ***Type matching rule*** *is to correctly invoke Windows API. Parameters from Java program must match the same size as native parameters (in C) and keep the same kind (primitive or pointer).*

API method signature is defined in a DLL proxy class only tells JNA how to copy parameters into the native stack. The native function will use the parameters in native stack by its original definition. The native instructions were compiled from C, therefore

input parameters are treated in same way they were defined in C. This is the reason why the size and kind of parameter must be kept.

In Java programs, each parameter need to have its type identification which enable it to satisfy principles in binary level. In Java programs, the parameters haves their type identification as "*byte, short, char, int, boolean, long, double, float*", then the value of them are stored in stack in binary level. The other parameters are objects whose value of them are stored in memory area pointed by a pointer. In the example of GetDateTime function, the type identification of parameters are *"int, int, SYSTEMTIME, WString, char[], int"*. Figure 2.3 shows the memory model for GetDateFormat call.

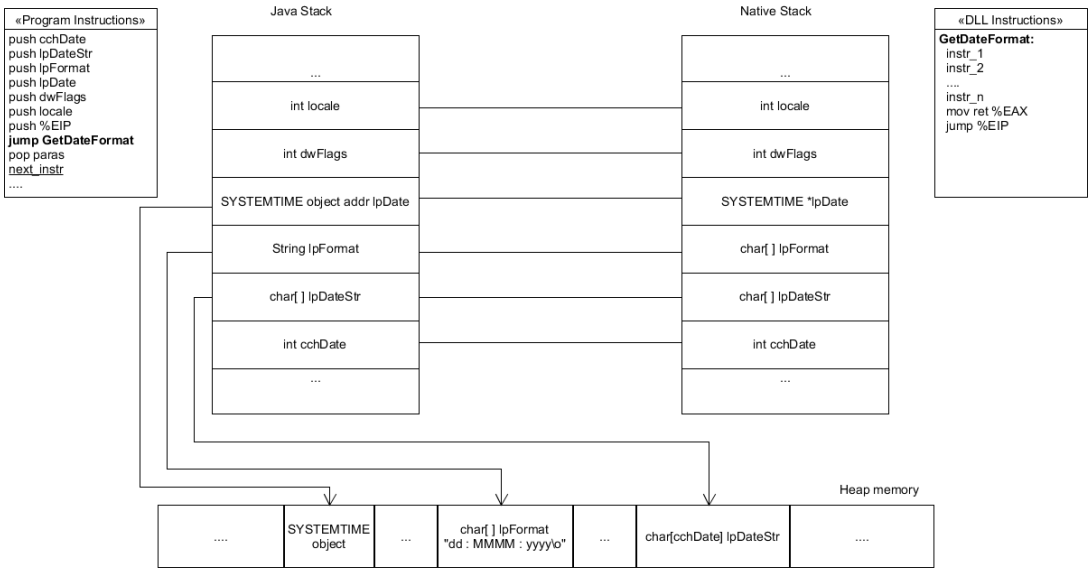


Figure 2.3: Memory model for GetDateFormat call via JNA

Figure 2.3 also shows assembly pseudo code, which stand for x86 assembly instructions in a real program. The left-hand side instructions stand for instructions in the program. Because GetDateFormat is called via JNA, each *push* instruction not only pushes a parameter into the Java stack but also copies it to the native stack. The *jump* instruction jumps to the first address of GetDateFormat instructions (outside the program). The next instructions of GetDateFormat are worked with the native stack. In the end of GetDateFormat function, the return value is stored in the *EAX* register, jump back the next instruction of main program by the value in *EPS* register. In summary, JNA lets compiler copy parameters into the native stack, and identify the address of API for the *jump* instruction.

2.3 Bayesian Learning and Sentence Similarity

We apply supervised machine learning and natural language processing techniques to extract API specification from its description.

- **Machine learning:** A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E [13]. The input of a machine learning algorithm is existing data, and the output is often a model (target function). The learning process is the execution of a program to optimize the function's parameters to fit training data.
  For example:
  - Task T: predicting the price of house in Kanazawa
  - Performance measure P: the difference between the real price and predicted price is lower than a threshold (e.g., lower than 10.000 Yen)
  - Training data E: data from 1000 houses in Kanazawa City over 10 years, data about each house includes house price, its area ($x \in N$), its age ($y \in N$) and the number of rooms ($z \in N$)
  - Target function: $Price : N^3 \rightarrow R$
  - Target function representation: $Price(x, y, z) = w_0 + w_1 x + w_2 y + w_3 z$ and parameters are w_0, w_1, w_2, w_3
- **Supervised learning** is the machine learning task of producing a inferrence function from labeled training data, and the function can be used to predict the class labels for unseen input instance [14]. The labeled training data includes a set of training examples, and each example is a pair of an input object (e.g., a feature vector) and a observed output value.
- **Natural language processing** are techniques to enable computer to interact with and natural languages. Computers traditionally only understand precise, unambiguous and highly structured language such as programming language; but natural language is often ambiguous in words, grammar structure and social context [11].

In the research, we compute sentence similarity to distinguish cell and buffer pointer. In addition, Bayesian learning is applied to predict memory length parameter (see 5.3).

Bayesian Learning

Bayesian Learning provides a probabilities inference approach to existing data by computing hypothesis's probabilities explicitly. [13]

- *Bayes Theorem*
$$P(h|D) = \frac{P(D|h)P(h)}{P(D)} \tag{2.1}$$
  D is the observed training data
  H is the hypothesis space, which is the set of all possible outputs.

[11] natural language processing (NLP)
`http://searchcontentmanagement.techtarget.com/definition/natural-language-processing-NLP`

$h(\in H)$ is a hypothesis (output)
$P(h)$ is the initial probability that hypothesis h holds, in supervised machine learning context $P(h)$ is the probability of h in the observing data D
$P(D)$ is the prior probability that training data D will be observed
$P(D|h)$ is the initial probability of observing data D given the hypothesis h
$P(h|D)$ is the probability of hypothesis h given the observing data D
With a new instance, the most probable hypothesis can be inferred from the observing data, by the maximum a posterior.

$$h_{MAP} = \underset{h \in H}{\operatorname{argmax}} P(h|D) = \underset{h \in H}{\operatorname{argmax}} \frac{P(D|h)P(h)}{P(D)} = \underset{h \in H}{\operatorname{argmax}} P(D|h)P(h) \tag{2.2}$$

$P(D)$ can be removed because it is constant by the training data and independent of h.

- *Naive Bayes Classifier* is the application of Bayesian learning to classification tasks. Instance X is described by a tuple of feature values $\langle a_1, a_2 \dots a_n \rangle$
the most probable output of X is

$$\begin{aligned} h^X_{MAP} &= \underset{h \in H}{\operatorname{argmax}} P(h|a_1, a_2 \dots a_n) \\ &= \underset{h \in H}{\operatorname{argmax}} \frac{P(a_1, a_2 \dots a_n|h)P(h)}{P(a_1, a_2 \dots a_n)} \\ &= \underset{h \in H}{\operatorname{argmax}} P(a_1, a_2 \dots a_n|h)P(h) \end{aligned} \tag{2.3}$$

To rewrite this formula, we assume that the attribute values are conditionally independent given the target value.

$$h^X_{MAP} = \underset{h \in H}{P}(h) \prod_i^n P(a_i|h) \tag{2.4}$$

For illustrative example, a restaurant try to predict whether the local children like a new candy or not. The data is collected in one year, and each data record is described by 4 features (shape, color, size, taste) and its label (like or dislike). The new candy has features: circle(shape), blue(color), big(size) and sweet(taste). From the data set, the likelihood for each class:
$P(like)P(circle|like)P(blue|like)P(big|like)P(sweet|like)$
$= 0.64 * 0.25 * 0.33 * 0.33 * 0.44 = 0.0077$
$P(dislike)P(circle|dislike)P(blue|dislike)P(big|dislike)P(sweet|dislike)$
$= 0.36 * 0.6 * 0.4 * 0.8 * 0.6 = 0.0414$
Furthermore, conversion into a probability by normalization:
$P(like) = 0.0077/(0.0077 + 0.0414) = 0.1568$
$P(dislike) = 0.0414/(0.0077 + 0.0414) = 0.8432$
We can conclude that the children don't like the new candy.

Cosine Similarity

- *Vector Space Model* is the presentation of text as a vector, and a vector comprises dimensions that are terms used to index it [15]. The definition and value of term depend on particular applications. For example, the vector model of a sentence:

$$s = (w_1, w_2 \dots w_n) \tag{2.5}$$

with s is a sentence which is presented as an high dimensional vector. Terms predefined keywords and values of terms are the weight of keywords. Keywords is a set of words: *"parameter, point, to, buffer"*. If the sentence is *"The parameter points to a buffer"*, the vector model of this sentence is $sent = (1, 3, 2, 3)$

- *Cosine similarity* is used as a measure of similarity between two vectors, and its value is the cosine of the angle between them [12] . Given two vectors $A = (a_1, a_2 \dots a_n)$ and $B = (b_1, b_2 \dots b_n)$

$$similarity(A, B) = cos(A, B) = \frac{A.B}{|A|.|B|} = \frac{\sum_{i=1}^{n} a_i.b_i}{\sqrt{\sum_{i=1}^{n} a_i^2}\sqrt{\sum_{i=1}^{n} b_i^2}} \tag{2.6}$$

Cosine similarity measure is a good approach to compute the similarity between two sentences, particular in cases that there is no big different between the expression of the same content. In the research, the value of sentence similarity is also used in two tasks: buffer or cell pointer classifier, and as a feature in predicting memory length parameter.

For example, the vector of "the parameter points to a buffer" is $[1, 3, 2, 3]$, and the vector of "pointer to a memory in which this function retrieves the formatted date string" is $[0, 3, 2, 3]$ (how to convert from sentence to its vector, see 5.3)

$$simil(sen1, sen2) = \frac{1.0 + 3.3 + 2.2 + 3.3}{\sqrt{1^2 + 3^2 + 2^2 + 3^2}.\sqrt{0^2 + 3^2 + 2^2 + 3^2}} = \frac{22}{\sqrt{23}.\sqrt{22}} \approx 0.98$$

[12] Cosine similarity
`https://en.wikipedia.org/wiki/Cosine_similarity`

Chapter 3

BE-PUM System

3.1 What is BE-PUM

BE-PUM (Binary Emulation for PUshdown Model generation) is a binary analyzer [12]. Currently, BE-PUM focuses on malware, which are often small and obfuscated. BE-PUM receives a x86/Win32 file binary executable and returns its control flow graph (CFG), which is constructed by symbolically execute the program on-the-fly manner.

BE-PUM adapts an on-the-fly-construction of CFG due to 2 reasons.

- In the binary level, there are no difference between data and instructions in memory. Therefore, instructions may be modified during the execution of the binary.
- The location of the next instruction is determined by both current instruction and environment (e.g., indirect jump eax, then the next instruction depends on the value in the eax register)

BE-PUM applies symbolic execution to execute programs. Symbolic execution is a technique to symbolically execute a program, and it maintains a symbolic state (l, pc) with l is the location of instruction and pc is the path condition from program entry to current instruction l [16]. The path condition pc expresses the precondition of the execution path to l, and if it is satisfiable, the execution path is feasible. In BE-PUM, the next instruction is decided by concolic testing (testing with a instance of pc). For stepwise executions, a virtual simulation is required.

BE-PUM restricts its binary emulator to the user process due to two reasons.

- Firstly, it allows BE-PUM to handle malware in a more flexible way, particular in trigger-based malwares. For example, a malware only executes on New Year's Day. In this case, the simulation of entire system (e.g., OllDbg or Intel/Pin) may fail to discover the real intended action, because the current date may not New Year's Day. BE-PUM gives choices on the return value of an API either symbolic or concrete. For instance, trigger-based behavior can be handled by symbolic execution, then discover the real behavior of malware.
- Secondly, in implementation aspect, it is heavy to implement a full simulation.

3.2 BE-PUM Architecture

BE-PUM is implemented on Java. Here, we borrow the figure and description of BE-PUM from [12] to briefly explain about BE-PUM architecture. It applies Jakstab 0.8.3 as a disassembler to determine a single-step disassembly, and SMT Z3.4.3 as the backend engine to generate a instance for concolic testing.

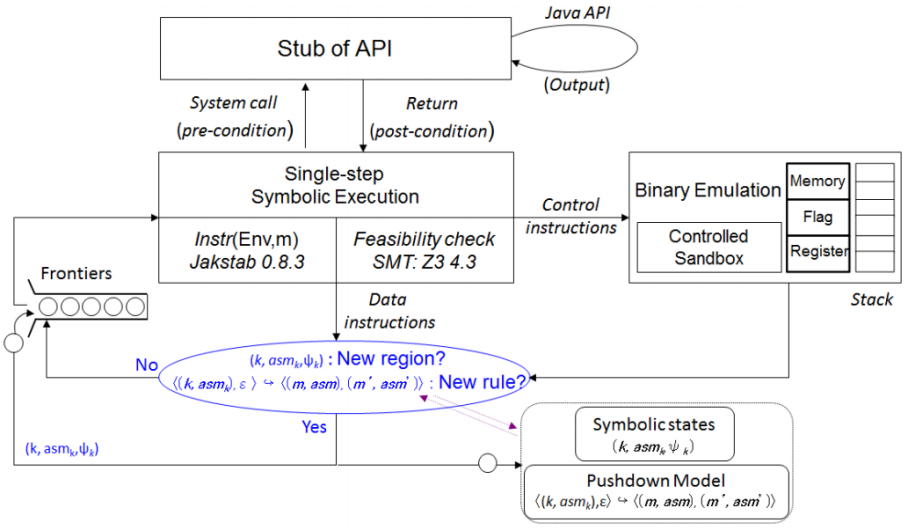


Figure 3.1: BE-PUM architecture

Figure 3.1 shows the architecture of BE-PUM with three components: symbolic execution, binary emulation and CFG storage. The symbolic execution picks up one from the frontiers (symbolic states at the ends of explored execution paths), and it tries to extend one step. If the instruction is a data instruction (i.e., only environment is updated and the next location is statically decided), it will disassemble the next instruction. If the instruction is a control instruction (e.g., conditional instruction jumps), the concolic testing is applied to decide the next location. After that, either a new CFG node or a new CFG edge is found, they are stored in CFG storage and a configuration is added to the frontiers. This procedure continues until either the exploration has converged, or coming to unknown instructions, system calls or addresses.

3.3 API Stub in BE-PUM

BE-PUM symbolically executes programs, and the path condition (on the symbolic value) and environment (the mapping between variables and their values) are separated in implementation. The environment (Env) is presented by a tuple (Env_R, Env_S, Env_M), with Env_R is the register values, Env_S is the stack values, Env_M is the memory values (excluding stack values). API Stub is used to handle system API calls. Each Windows API call is treated as a single instruction, and update the environments following the

technical document from MSDN. However, BE-PUM keeps the path condition based on observation (see 4.1).

Stub is the piece of code simulating the behavior of other code [1]. In BE-PUM implementation, API stub is a proxy object that can invoke API native function and update the simulation environment after API call. It makes flexibility in symbolic execution, and avoids the cost of manual APIs approximation. Current BE-PUM covers 400 APIs, which were implemented manually. However, there are about more than 4000 published APIs, and the lack of APIs stub may lead the unexpected termination of BE-PUM. Therefore, we aim to automatically generate Windows APIs Stub in order to reduce manual effort and increase the ability of BE-PUM.

The flow process of API stub is described through GetDateTime stub. Figure 3.2 shows the flow process of GetDateTime call.

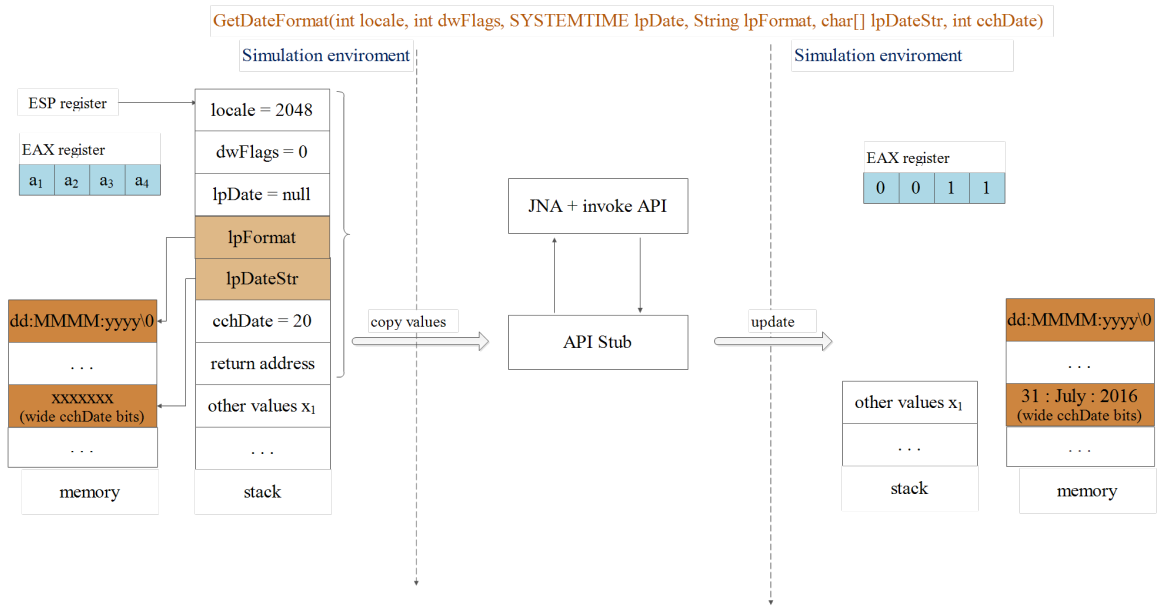


Figure 3.2: GetDateFormat Flow Process

The flow of an API stub can be divided into 5 stages.

- Based on the number of parameters, the values are popped from the stack of simulation environment to variables in API Stub program.
- Depending on the API specification, if the parameter is a pointer, the values of the pointed memory area also copied to variables in API stub program.
- In the API stub, JNA passes the variables as input parameters to the native stack in the real environment and invoke API native function.

[1] What is a Stub?
`http://www.tutorialspoint.com/software_testing_dictionary/stub.htm`

- JNA comes back Java stack and converts results back to appropriate variables in API stub program.
- Depending on the API specification, API stub copies the value of variables to appropriate EAX register and Memory objects in the simulation environment.

Almost Windows API and kernel-level structures are implemented and compiled from C programming code[2]. The environment after Windows API call is updated as following. On x86 platforms, after a Windows API call:

1. The return value is always widened to 32 bits and stored in the EAX register [3]. For example, the return type can be int, long (32 bits in x86 platform), boolean, a structure wide 32 bits (e.g. COORD structure), pointer to a memory address, or void (no return value, the value of eax register is kept as before API call).

   For example, as GetDateFormat function and parameters figure 3.2, the return value is $0x0011_{16}(= 17_{10})$ stored in EAX register after GetDateFormat call.

2. The value of memory area pointed by a pointer parameter can be updated. Parameters are passed by value to Windows API function [4]. The copies of actual parameters are passed into a function, therefore the changes of parameters inside the function have no effect on actual parameters. However, even the value of pointer parameter doesn't change, the memory area are pointed by a pointer parameter can be modified.

   Furthermore, Microsoft source-code annotation language (SAL) is a set of annotations to describe how a function uses its parameters [5]. The SAL of a parameter is described in the function prototype of API description. If SAL contains "Out", such as "_Out_", the function may write to the memory area pointed by pointer parameters.

   For example:

```
// Function prototype
int GetDateFormat(
  _In_            LCID       Locale,
  _In_            DWORD      dwFlags,
  _In_opt_  const SYSTEMTIME *lpDate,
  _In_opt_        LPCTSTR    lpFormat,
  _Out_opt_       LPTSTR     lpDateStr, // can modify memory area
```

[2]Windows Programming/C and Win32 API
https://en.wikibooks.org/wiki/Windows_Programming/C_and_Win32_API
[3]Argument Passing and Naming Conventions
https://msdn.microsoft.com/en-us/library/984x0h58.aspx
[4]__stdcall Calling Conventions
https://msdn.microsoft.com/en-us/library/zxk0tw93.aspx
[5]SAL Annotations
https://msdn.microsoft.com/en-us/library/ms235402(v=vs.100).aspx

```
  _In_            int         cchDate
);
```

The memory area pointed by parameter "*lpDate, lpFormat, lpDateStr*" can be modified. However, from SAL notations we see that only the SAL of parameter "*lpDateStr*" contains "Out", then we know that memory area pointed by "*lpDateStr*" may be updated after API call.

3. A new memory area is created and pointed by the return value. This situation can occur by several techniques in C language, such as local static variable and dynamic memory allocation techniques. We observe that such APIs are few, and this specification requires deep understanding from natural description. We leave this specification for future work, and assume the return value of API doesn't point to a new memory area.

Chapter 4

Observation

The our observations to decide our implementation choice are presented.

4.1 Stub and Path Condition

In Symbolic execution, the value of program variables are presented as symbolic values or symbolic expressions instead of concrete input values [16]. For example, the value of variable x is presented by a symbolic expression "$\alpha + 10$", and α is a symbolic constant. The collection of all variables and their values is called a environment. The logical constraints on symbolic value from the entry of program to a specific instruction is called a path condition. Both environment and path conditions are updated along an execution path of the program. In theory, in each step execution both environment and symbolic path constraint need to be maintain together, in other words the environment need to keeping in each step.

In the implementation of symbolic execution, the path conditions and the environment often separated to reduce memory space and easier in updating environment. BE-PUM also implements symbolic execution by separating path condition and environment.

When a program uses a stub to invoke a external function, we often know only input-output and do not know the body of function. Once, an API is invoked, in normal behavior there is no conditional branch based on symbolic values and its external action is symbolic value. Except that, inputs causes errors, which lead exceptions. Particularly in malware, we assume that input parameters for invoking API are correct. Therefore, the stub only updates the environment, and the patch condition is kept.

For example of stub in real applications:

- Symbolic execution of programs uses SQL stubs to invoke query functions in mySQL database that is outside of the Java programs.
- Symbolic execution of a mobile applications uses Google Map API stubs to invoke functions in servers of Google that are outside of the application.

- In the context of research, symbolic execution of malwares uses Windows API stubs to invoke Windows API functions in DLL files that are outside of the programs.

4.2 Required Windows API Specification

Each API function often contains various specifications, but we do not need to all of them in the context of BE-PUM system. The required specifications are following.

- To implement API stub, we need to invoke API via JNA and update the environment in BE-PUM, which requires the specification for API stub generation.
- To verify generated API stub, conformance testing needs to be applied. The idea is to compare the memory values between a real simulator (e.g., Intel/Pin and OllDbg) and BE-PUM. To do that, test programs containing target API call need to be generated, which requires the specification for test case generation.

The required specification for Windows API stub generation is the information how to initialize input values before an API call and how to update the environment after an API call. For then, we need:

- The library (DLL) that an API belong to
- The function name of an API
- The numbers of input parameters
- The parameter type identification in Java and its definition (for structure only)
- The name of the parameter (it can be optional, but the original name of parameter should be kept, which make generated program become friendly with developers)
- The memory length parameter of a buffer pointer.

The required specification for test case generation further requires two specifications as following.

- *Conditions for the validity of input parameters*: In a test case program, the input parameters must have initial values before calling API function. However, the input executable files to BE-PUM are malware programs, which usually can call system API successfully. Therefore, in API stub generation the condition for valid input can be ignored, but in test case generation it is required.
- *The dependence between Windows APIs*: Many APIs require input values that are return values of another API. The values are often given by operating system at running time. In an API stub generation, the dependence between Windows API can be ignored, because the return value of preceding API have already stored in the simulation environment. However, in test case generation the dependence specification is required.

Extracting specifications for test case generation in conformance testing purpose is more difficult, since the required specifications are embedded more deeply in natural language description. They are left for future works.

4.3 Description Format of Windows API Specification

The Windows API specifications are described in Windows API description. Wee see the description formats with the example of GetDateFormat function.

Specifications for API stub generation

API requirements is described in table format, which also includes the information about DLL. Figure 4.1 shows the requirement description for GetDateFormat call.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Minimum supported phone	Windows Phone 8
Header	Datetimeapi.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

Figure 4.1: Requirement description in GetDateFormat

API function prototype is described in C. From the description, API function name, the number of parameters, type identification of parameters, the name of parameters can be obtained. Fortunately, Microsoft provides type name conventions [1] . For example, as coding convention the prefix P- or LP- stands for pointer type (e.g., LPRECT is a pointer to a RECT structure, LPRECT and RECT* are the same). The postfix -STR stands for string types (pointers at binary level). Under such conventions, the parameter is whether pointer or not can be inferred.

For instance, figure 4.2 shows that "*lpDate, lpFormat and lpDateStr*" are pointer parameters.

The parameter description is described in natural language. From the description, the type definition (may link to other documents), the relation between parameters are obtained. The description of *"lpDateStr"* is *"Pointer to a buffer in which this function retrieves the formatted date string"*, which means *"lpDateStr"* parameter is a buffer

[1]Windows Coding Conventions
`https://msdn.microsoft.com/en-us/library/windows/desktop/ff381404(v=vs.85).aspx`

Syntax

C++

```
int GetDateFormat(
  _In_            LCID       Locale,
  _In_            DWORD      dwFlags,
  _In_opt_  const SYSTEMTIME *lpDate,
  _In_opt_        LPCTSTR    lpFormat,
  _Out_opt_       LPTSTR     lpDateStr,
  _In_            int        cchDate
);
```

Figure 4.2: Prototype description of GetDateFormat

Parameters

Locale [in]

Locale identifier that specifies the locale this function formats the date string for. You can use the MAKELCID macro to create a locale identifier or use one of the following predefined values.

- LOCALE_CUSTOM_DEFAULT
- LOCALE_CUSTOM_UI_DEFAULT
- LOCALE_CUSTOM_UNSPECIFIED
- LOCALE_INVARIANT
- LOCALE_SYSTEM_DEFAULT
- LOCALE_USER_DEFAULT

dwFlags [in]

Flags specifying date format options. For detailed definitions, see the *dwFlags* parameter of **GetDateFormatEx**.

lpDate [in, optional]

Pointer to a SYSTEMTIME structure that contains the date information to format. The application sets this parameter to **NULL** if the function is to use the current local system date.

lpFormat [in, optional]

Pointer to a format picture string that is used to form the date. Possible values for the format picture string are defined in Day, Month, Year, and Era Format Pictures.

The function uses the specified locale only for information not specified in the format picture string, for example, the day and month names for the locale. The application can set this parameter to **NULL** to format the string according to the date format for the specified locale.

lpDateStr [out, optional]

Pointer to a buffer in which this function retrieves the formatted date string.

cchDate [in]

Size, in characters, of the *lpDateStr* buffer. The application can set this parameter to 0 to return the buffer size required to hold the formatted date string. In this case, the buffer indicated by *lpDateStr* is not used.

Figure 4.3: Parameter description in GetDateFormat

pointer. The description of *"cchDate"* is *"Size, in characters, of the lpDateStr buffer"*, which means the length of memory pointed parameter *"lpDateStr"*.

Pointer of pointer parameter can be inferred from type name convention. For example, the prefix P-* or LP-* stands for pointer of pointer type (e.g., LPINT* is a pointer of int pointer, LPINT* and int** are the same). The stub generation system recognizes such cases and copies the appropriate memory values in BE-PUM to API stub program. We also observe that the number of APIs have pointer of pointer parameter is small, and the maximum is 2 deep level.

Nested structure is the structure that has other structures inside it. To handle these cases, the stub generation applies a recursive process until all fields of structure is primitives. We observe that the maximum deep of nested structure is three, and these cases are very rare. The most frequency is is non-nested structure.

Specifications for test cases generation

The parameter description is described in natural language. The condition for parameters are obtained from the description. Depending on API, the conditions for valid inputs are whether required or not. If an parameter has conditions, which part, sentence and format need to be determined. By observation, there are several formats used to describe the condition, such as natural language, table (various table formats), enum (can be linked to other web-page). Besides, the number of possible conditions are also various. Figure 4.4 shows that possible values for parameter *"Locale"* described in the other pages.

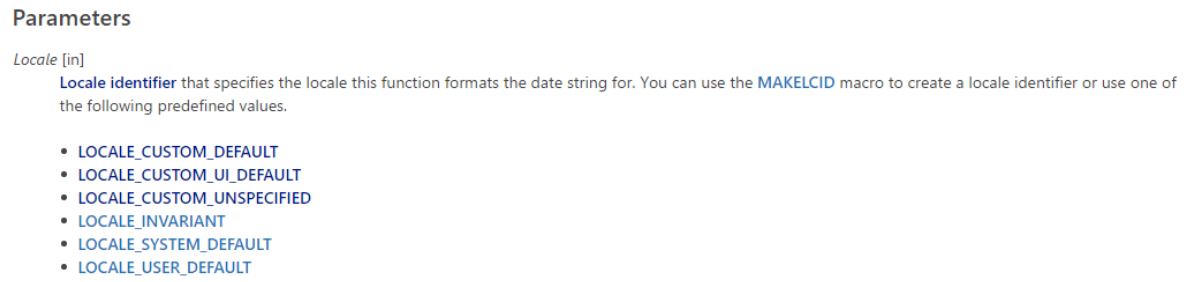


Figure 4.4: The description of parameter Locale in GetDateFormat

The API's explanation is described in the natural language. From the description, the dependence between APIs can be obtained. Figure **??** shows that the value of

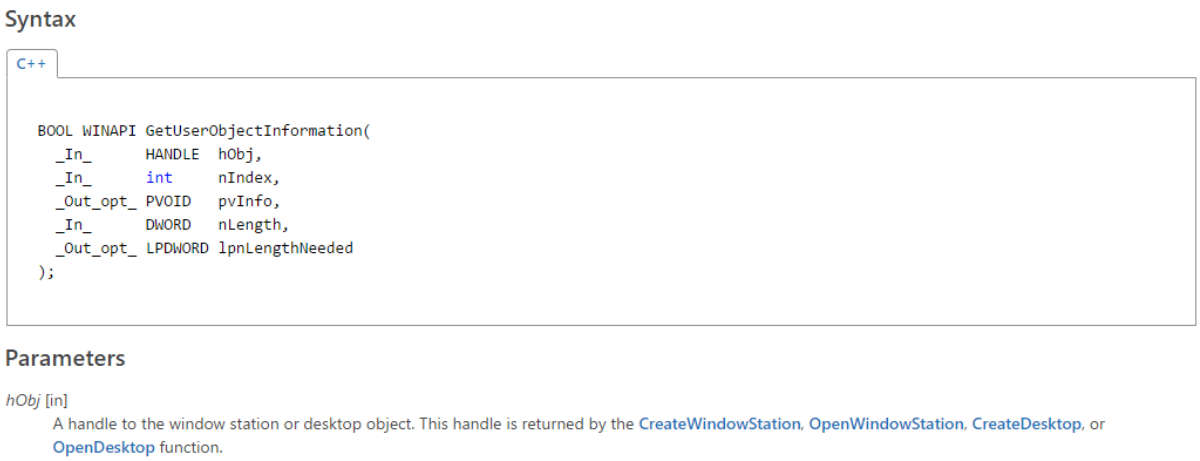


Figure 4.5: A part of the GetUserObjectInformation description

parameter "*hObj*" should be obtained from other APIs, such as CreateWindowStation, OpenWindowStation, CreateDesktop, or OpenDesktop function.

Chapter 5

Specification Extraction

5.1 API Identification

To map to a native API function via JNA, the library which API belong to, the function name of API and the number of input parameters need to be identified. Such specifications can be determined through the description of requirements and API prototype.

The library (DLL) which API belong to is determined through the description of requirement. The requirement description is in table format, and it is often the last table in an API document. The name of library is be extracted from the second cell in the row DLL. The example of requirement description of GetDateFormat is showed as following.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Minimum supported phone	Windows Phone 8
Header	Datetimeapi.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

In this example, “kernel32” is extracted from the sixth row.

The function name of API and the number of input parameters are determined through the description of API prototype. Because API prototype is described in C, the information can be extracted easily. The API name is always the last word in the first line, the number of parameters are the number of comma symbol plus one. For example, GetDateFormat prototype is showed as following, and the name of API is “GetDateFormat”, and there are six input parameters.

Syntax

C++

```
int GetDateFormat(
  _In_            LCID       Locale,
  _In_            DWORD      dwFlags,
  _In_opt_  const SYSTEMTIME *lpDate,
  _In_opt_        LPCTSTR    lpFormat,
  _Out_opt_       LPTSTR     lpDateStr,
  _In_            int        cchDate
);
```

5.2 API Parameters

As mentioned in 2.2, passing parameters from Java stack to the native stack is taken by JNA. However, to correctly invoke Windows API, parameters from Java program must match the same size as native parameters (in C language) and keep the same kind of parameter (primitive or pointer). Therefore, in Java programs each parameter need to have its type identification which enable it to satisfy *type matching rule* in 2.2. The parameter's type identification in Java can be inferred from the parameter's type identification in C by coding convention.

If the value of parameter is stored in stack, it is called a *primitives parameter*. The parameter type identification in Java corresponding to the parameter type identification are showed as below table [1].

Windows Type identification	Size	C primitive type	Java primitive type
BYTE, TCHAR	8 bits	char	byte
WORD, TCHAR	16 bits	short, wchar_t	short, char
DWORD, BOOL, LONG	32 bits	int, long	int, boolean, NativeLong
__int64	64 bits	long long	long, double
float	32 bits	float	float
double	64 bits	double	double

If the real values of parameter is a memory area pointed by a pointer, the parameter is called a pointer parameter. To easier analysis, we divide pointer parameter into several kinds.

- **Structure pointer:** A structure in Java need to be corresponded to a structure in C. The structure in Java is an object, which is a pointer at binary level. Furthermore, the memory size of object must be match the memory size of structure. To satisfy this requirement,the class that is equivalent with original structure should be defined. To allow JNA to process such kind of pointer correctly, the class for structure must be extended from class *jna.Structure*, and all fields of class need matching all fields of structure by principle rules.

[1] https://github.com/java-native-access/jna/blob/master/www/Mappings.md

⟨*Problems*⟩ JNA supports some common classes for Windows API's structures, however there remain many undefined structures.

⟨*Solution*⟩ We collect structure description, and generate corresponding classes.

- **Cell or buffer pointer**: A pointer variable can point to either only one element (cell pointer) or sequence of elements (buffer pointer).
  ⟨*Problems*⟩ From API prototype, we don not know this information.

  ⟨*Solution*⟩ We apply cosine similarity between the first sentence of parameter description and the base sentence *"the parameter points to a buffer"* in order to classify cell pointer or buffer pointer.

- **Array and number of elements**: Array is a sequence of variables. To pass array to a function, we need to pass a pointer pointing to the first element of array and the number of elements in array.

  ⟨*Problems*⟩ Based on parameter description in API description, we need to know which parameter describes the number of elements in an array. In Windows API, the immediate parameter after a buffer pointer parameter isn't necessary to be the number of elements in an array.

  ⟨*Solution*⟩ We apply cosine similarity between the first sentence of parameter description and the base sentence *"the parameter points to a buffer"* in order to classify such cases.

- **String** is an array of characters, and is terminated by null character "/0". Passing String to a function doesn't require the length of character array because compiler can handle the termination of string by character "/0".

  ⟨*Problems*⟩ From function prototype, we don't known a parameter is a String or array of characters, because both of them are defined as "*char* or char[]*".

  ⟨*Solution*⟩ To distinguish such cases, the research applies cosine similarity to estimate the similarity of the first sentence to a base sentence *"the parameter points to a buffer"*.

The names of parameters are optional, and are kept for future convenience.

5.3 Buffer Pointer and Memory Length Parameter

As mention in the previous section, we apply sentence similarity to classify cell or buffer pointer, and Naive Bayes classifier to predict memory length parameter. They are presented in detail in this section.

Sentence similarity

In the research, the similarity of two sentences is the cosine similarity between their sentence vectors (see formula 2.6 in 2.3). The sentence similarity is used in 2 tasks: cell or buffer pointer classifier, and as a feature in predicting memory length parameter.

- To classify cell or buffer pointer, we observe that we do not need entire parameter description. We often understand whether cell or buffer pointer based on the first sentence of parameter description. Therefore, the idea is computing the similarity of the first sentence with a base sentence "*the parameter points to a buffer*" to classify whether cell or buffer pointer. The base sentence is manually designed. If a parameter is a pointer and its similarity is greater than 0.8, the parameter is predicted as a buffer pointer.

- To predict memory length parameter, we also can understand whether cell or buffer pointer based on the first sentence of parameter description. Therefore, the idea is computing the similarity of the first sentence with a base sentence "*the parameter describes the length of buffer*", and the base sentence is manually designed. However, if predicting memory length parameter bases on only the sentence similarity, the accurateness is not high by experiments. Therefore, we use the additional information of parameter to increase the accurateness, which is discussed in more detail in the part Naive Bayesian binary classifier.

The key point in computing sentence similarity is how to transfer from a sentence to its vector, then the similarity is straightforward computation based on the formula 2.6.

- The sentence is presented by a n-dimensional vector $(w_1, w_2 \dots w_n)$. In this research, the number of dimension is equal to the number of keywords for a specific tasks.

  - In cell or buffer pointer classifier task, a sentence vector has 4 dimensions, and the keywords include "*parameter, point, to, buffer*".
  - In memory length parameter task, a sentence vector has 5 dimensions, and the keywords include "*parameter, describe, length, of, buffer*".

  In NLP, the similarity between two sentence can be computed by semantic parsing technique. However, due to the lack of dictionary for software area and technical words, this technique requires more effort. In vector model, pre-design keywords implicitly determine the meaning of a sentence for a specific task. From experiments, our approach requires less effort, but good efficiency.

- In sentence vector, each term "$w_1, w_2 \dots w_n$" has a value corresponding to the weight of a keyword. In other word, if a keyword appears in the sentence, the value of term corresponding to this keyword is the weight of this keyword.

  - In cell or buffer pointer classifier task, the weight of keywords as below.

Keyword	parameter	point	to	buffer
Weight	1	3	2	3

– In memory length parameter task, the weight of keywords as below.

Keyword	parameter	describe	length	of	buffer
Weight	1	1	3	2	3

The weight of keywords is manually designed by observation about the important of each keyword. In some other research contents, such as document similarity, the value of term can be the multiply of its weight and its frequency. However, in our research context, only one sentence is considered and the length of sentence is short, then keywords often do not appear more than one time. In addition, we consider the similarity between one description sentence with a designed base sentence, instead of the similarity between arbitrary sentence, and word frequency reduces the correctness because every keywords appear only one time in the base sentence.

- The keywords can be expressed by their synonyms in the sentence. If the synonym of a keyword appears in the sentence, it is equal to the appearance of this keyword. The synonyms can be defined automatically by a dictionary (e.g., WordNet), but due to the narrowness of research domain, the approach of pre-defined synonyms is more efficient. The below table shows keywords and their synonyms.

Keyword	Synonyms
parameter	argument, the specific name of parameter
point	points, pointer
describe	describes, express, expresses, present, presents
length	size, amount
buffer	memory, array

An example of converting sentence to its vector in cell or buffer pointer classifier task is showed as below.

```
# ----- Base sentence
sent1 = '''The parameter points to a buffer'''
# keywords appear in sent1 are "parameter, points, to, buffer"
vect1 = [1, 3, 2, 3]

# ----- Description of parameter lpDateStr
sent2 = '''Pointer to a buffer in which this function retrieves the
        formatted date string.'''
# keywords appear sent2 are "pointer, to, buffer"
vect2 = [0, 3, 2, 3]

# ----- Description of parameter lpDate
```

```
sent3 = '''Pointer to a SYSTEMTIME structure that contains the date
        information to format.'''
# keywords appear sent3 are "pointer, to"
vect2 = [0, 3, 2, 0]

# Apply formula 2.6
simil(sent1,sent2) = 0.98; simila(sen1,sent3) = 0.75
# Therefore, lpDateStr is predicted as a buffer pointer.
```

Naive Bayesian binary classifier

When the buffer pointer parameter is determined in an API, we apply Naive Bayes learning to classify whether a parameter is memory length parameter or not. As mention, memory length parameter describes the length of buffer pointed by a buffer pointer.

As mention in 5.3, each parameter description need to be converted to a tuple of features. In this task, there are 5 features.

- The first is the feature about parameter SAL annotation, and the possible values of this feature are "Yes, No, Unknown". If SAL annotation contains "Out", then the value of this feature is "Yes". If SAL annotation contains "In", then the value of this feature is "No". Otherwise, the value of this feature is "Unknown".
- The second is the type's identification of parameter, and the value of this feature is a string. For example, *int, long*
- The third is whether the parameter name contains keywords "*length, size*" or not, and the possible values of this feature are "Yes, No".
- The fourth is the distance from buffer pointer parameter, and the value of this feature is a integer number. This number is the subtraction between the index of buffer pointer parameter and current parameter.
- The final is whether the sentence similarity is greater than 0.8 or not,and the possible values of this feature are "Yes, No". The sentence similarity is computed as the method in the previous part.

By experiment, the third and fifth feature contributes much to the accurateness of algorithm.

For example, the features of parameter cchDate in GetDateFormat function are extracted as below example.

```
# _In_ int cchDate (parameter description in prototype)
featr1 = Yes   # because SAL = '_In_'
featr2 = 'int' # because the type of this parameter is 'int'
featr3 = No    # because the name 'cchDate' does not contain keywords
featr4 = 1     # because buffer pointer is the fifth parameter, and
```

```
                #   and current parameter is the sixth.

# base sentence = 'parameter describes the size of buffer'
# parameter sentence = 'Size, in characters, of the lpDateStr buffer'
featr5 = Yes   # because cosine similarity = 0.95
```

In the research, the training data including the labeled description of 122 parameters are manually prepared. The label of parameter description is "yes"(memory length parameter) or "no"(not memory length parameter). Training process is computing all conditional probability of feature values in given outputs (see formula 2.3). For example, a part of model as below table, and each cell contains the probability of $P(featr|output)$

Feature values	output = yes	output = no
featr1 = yes	0.377	0.339
featr2 = 'int'	0.150	0.0188
featr3 = no	0.113	0.603
featr4 = 1	0.283	0.037
featr5 = yes	0.320	0.012

Besides, from training data $P(output = yes) = 0.396$ and $P(output = no) = 0.622$. For instance, the first cell contains 0.3.77, which means the proportion of the number of parameters whose SAL contains "In" among memory length parameters.

To predict whether a parameter is a memory length parameter or not from its description. Firstly, we need to convert its description to a tuple of features, then apply formula 2.3 to compute the probability of output. For example, the probability of parameter "cchDate" is whether memory length parameter or not as below.

- $P(yes|yes, int, no, 1, yes) = 0.000229162025664/P(yes, int, no, 1, yes)$
- $P(no|yes, int, no, 1, yes) = 0.000001061324560/P(yes, int, no, 1, yes)$

Therefore, the probability of memory length parameter is 99.5%; and the probability of not memory length parameter is 0.05% after normalization. We can predict that the parameter "cchDate" is the memory length parameter. In addition, if there are more than one predicted memory length parameter, we choose the one which has the highest probability.

Chapter 6

Stub Generation

In this chapter, the three kinds of generated code needed for API stub are going to presented. They are classes corresponding to structures, library proxy interface, and APIs stub.

6.1 Class for Structure Definition

The purpose of the class for structure is to let JNA understand this type as a Structure in C. The class must satisfy the following conditions.

- The class is derived from the class jna.Structure.
- All fields of the class in Java need to match all fields of the structure in C by *type matching rule* (in 2.2). If a structure is a nested structure, other classes of sub-structure also need to be generated. Another approach is the use of inner class, but it is more complex in implementation.
- The name of fields is kept as original name (optional).
- Method getFieldOrder() is override. This method return a list of field names. It is the requirement from the design of JNA.

For example, SYSTEMTIME defines an structure for date and time, the definition of structure SYSTEMTIME taken from MSDN is showed as below [1]

```
// Struct SYSTEMTIME definition in C language
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
```

[1] SYSTEMTIME structure
https://msdn.microsoft.com/en-us/library/windows/desktop/ms724950(v=vs.85).aspx

```
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME;
```

The SYSTEMTIME class is defined in Java as below.

```
// Class SYSTEMTIME in Java language
public class SYSTEMTIME extends Structure {
  public short wYear;
  public short wMonth;
  public short wDayOfWeek;
  public short wDay;
  public short wHour;
  public short wMinute;
  public short wSecond;
  public short wMilliseconds;

  @Override
  protected List<String> getFieldOrder() {
    return Arrays.asList(new String[] {
      "wYear", "wMonth", "wDayOfWeek", "wDay",
      "wHour", "wMinute", "wSecond", "wMilliseconds"});
    }
}
```

6.2 Interface for DLL proxy

The purpose of DLL proxy is to create a single access point to each file DLL by a JNA proxy object. Interface DLL proxy must satisfy several following conditions:

- The interface is derived from jna.win32.StdCallLibrary.
- The DLL and function's name is kept as original name.
- Parameters satisfy the type matching rule (see 2.2)
- The names of fields are kept as original name (optional).

For example, Kernel32.dll proxy interface is defined as below.

```
public interface Kernel32DLL extends StdCallLibrary {
        Kernel32DLL INSTANCE = (Kernel32DLL)
                Native.loadLibrary("kernel32",Kernel32DLL.class);
```

```
        // API functions
        public int GetDateFormat(int Locale, int i, SYSTEMTIME lpDate,
                String lpFormat, char[] lpDateStr, int cchDate );
        // .....
}
```

6.3 API Stub Class

API stub class is the core of generation task. In the constructor method, the number of parameter is defined in a constant NUM_OF_PARMS. The method execute() plays central role and handle four responsibilities.

- Getting original parameter values from stack and memory in BE-PUM.
- Coping the values of memory area in BE-PUM if parameters are structure pointer, or strings or array. Then input parameter objects are instantiated.
- Invoking API function
- Updating the environment in BE-PUM by return value and updated value of parameters.

For example, GetDateFormat Stub class is defined as below.

```
public class GetDateFormat extends Kernel32API {
  public GetDateFormat () {
    super();
    NUM_OF_PARMS = 6;
  }

  @Override
  public void execute() {
    // Step 1: get original parameter values from stack
    long t0 = this.params.get(0);
    long t1 = this.params.get(1);
    long t2 = this.params.get(2);
    long t3 = this.params.get(3);
    long t4 = this.params.get(4);
    long t5 = this.params.get(5);

    // Step 2: instantiate input parameters
    LCID Locale = new LCID (t0);
    DWORD dwFlags = new DWORD (t1);
    SYSTEMTIME lpDate = null;
    if (t2 != 0L) {
      lpDate = new SYSTEMTIME ();
```

```
      lpDate.wYear = (short) ((LongValue)memory.getWordMemoryValue
              (t2)).getValue();
      lpDate.wMonth = (short) ((LongValue)memory.getWordMemoryValue
              (t2+=2)).getValue();
      lpDate.wDayOfWeek = (short)
              ((LongValue)memory.getWordMemoryValue (t2+=2)).getValue();
      lpDate.wDay = (short) ((LongValue)memory.getWordMemoryValue
              (t2+=2)).getValue();
      lpDate.wHour = (short) ((LongValue)memory.getWordMemoryValue
              (t2+=2)).getValue();
      lpDate.wMinute = (short) ((LongValue)memory.getWordMemoryValue
              (t2+=2)).getValue();
      lpDate.wSecond = (short) ((LongValue)memory.getWordMemoryValue
              (t2+=2)).getValue();
      lpDate.wMilliseconds = (short) ((LongValue)memory.getWordMemoryValue
              (t2+=2)).getValue();
    }
    String lpFormat = null;
    if (t3 != 0L) lpFormat = memory.getText(this, t3);
    char[] lpDateStr  = null;
    if (t4 != 0L) lpDateStr = new char[(int) t5];
    int cchDate = (int) t5;

    // Step 3: call API function
    int ret = Kernel32DLL.INSTANCE.GetDateFormat (Locale, dwFlags,
            lpDate, lpFormat, lpDateStr, cchDate);

    // Step 4: update environment (memory & eax register)
    long value = ret;
    register.mov("eax", new LongValue(value));
    memory.setText(this, t4, new String(lpDateStr));
  }
}
```

Chapter 7

Implementation

The implementation of APIs stub generation is written by Python, Java and Velocity script language in about 6000 lines. The current system divided into 3 modules: crawler, extractor and generator. The conformance testing module is felt for future work.

7.1 Module Collector

The Windows API and structure description are published officially in the website of Microsoft Developer Network (MSDN). However, the collection of document is not prepared beforehand, therefore data preparation and pre-processing stage are required.

Windows API documents

- MSDN website provides several pages, which contain a list of Windows API document addresses [1] [2]. By observation, the list of APIs in such pages are not the same, and there also exist API that are not listed. Even API descriptions exists somewhere in MSDN website, collecting all published documents is not easy. There are also Windows APIs whose document are left unpublished, and we ignore such APIs.

- *Current solution* is starting from the web-page that contains the largest number of APIs. The idea is applying web crawling technique (e.g., web auto navigation and via Google Search API) to collect API documents, and pre-processing data technique (e.g., data cleaning and data transformation) to remove noise and transform documents into appropriate forms for analysis.

- *Further solution (future works)* is crawling from multiple sources, and then data integration approach is applied to combine/remove duplicate documents.

[1] Windows API Index
`https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx`

[2] Functions in Alphabetical Order
`https://msdn.microsoft.com/en-us/library/aa383688%28VS.85%29.aspx`

Structure definition document

- The document for structure descriptions are collected from the structure's URL address in API's parameter explanation of API description. However, this approach may fail due to missing the link between structure's document and API's document. There are many structure definitions do not have their document published in MSDN website.

- *Further solution is* using header files in Windows Software Development Kits (Windows SDK). The header files contains nearly complete Windows API prototypes, type definition and constants. However, the number of header files are very big, thousands of header files, millions of codes line and additional notation compared to original C (the exactly quantity depends on version of SDK). To obtain all information from these files, the program which is similar to a compiler is required. Moreover, head files contain API prototypes, but not other API descriptions.

In summary, this module takes three responsibilities: collecting Window's API description, collecting structure description and cleaning noise data. The descriptions are stored offline as HTML format files. We collected 1802 Windows API descriptions, 496 structure descriptions and 1 description for renaming primitive types [3]. This module is implemented in Python language.

7.2 Module extractor

The aim of this module is to extract API information from HTML document (semi-structure) and stored them into an structure format (hierarchical relationship format), which is implemented as a nest-diction in Python.

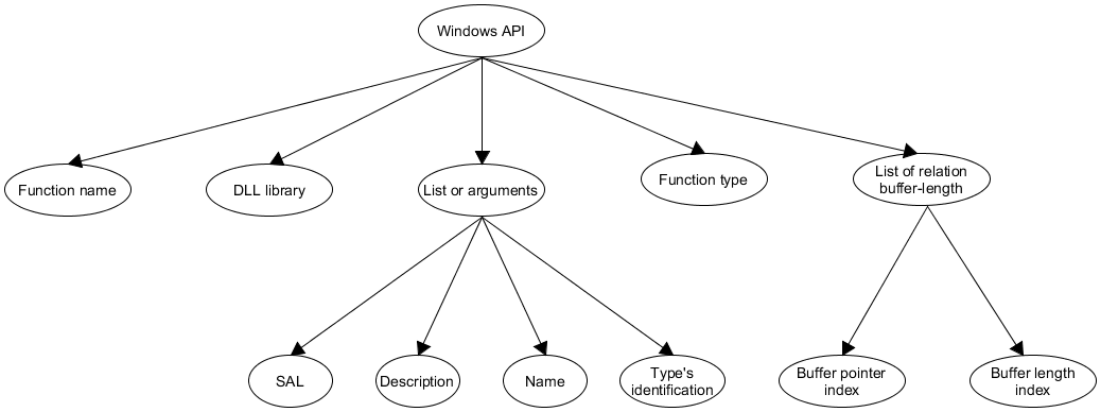


Figure 7.1: API Information Format

[3]Windows Data Types
`https://msdn.microsoft.com/en-us/library/windows/desktop/aa383751(v=vs.85).aspx`

Regular expression is used to extract DLL library's name, API name, parameter's name, parameter's SAL, parameter's description.

Parameter type identification in Java is inferred from the parameter identification in C and rules constructed by coding convention from MSDN. The rules are extracted from explicit rules for primitives (e.g., #define long LONG) and implicit rules for pointers (prefix P- or LP- stands for pointer).
For example,

- long = LONG = DWORD = LCID = LCTYPE = COLORREF = ...
- int* = INT* = PINT = LPINT = ...

In implementation, for analyzing easier and keeping consistency preservation, a weighted directed graph to express relation between type identification is implemented.

Figure 7.2 shows a part of the graph.

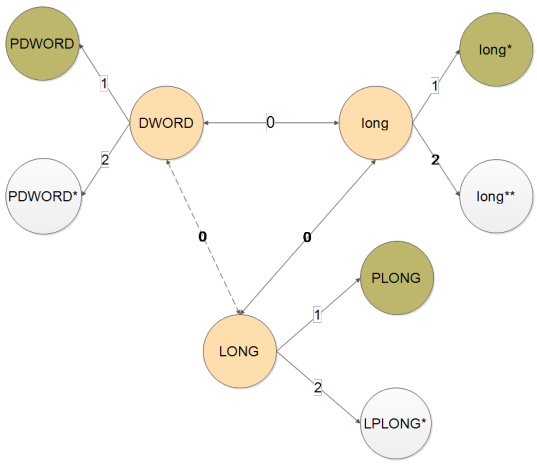


Figure 7.2: Sample of type graph

Pointer parameter classifier: The parameter is determined whether or not based on parameter type identification in C. If the type's identification is "P-, LP-, -*", the parameter is a pointer. To decide whether pointer parameter is buffer pointer, the cosine similarity is used. We compute the similarity of the first sentence in parameter description with base sentence " the parameter points to a buffer". If the similarity is greater than 0.83, the pointer type is considered as a buffer pointer.

Memory length parameter classifier: Naive Bayes learning is used to build a model from 122 parameter instances with 5 features (see 5.3). Training data are prepared manually. We observe that the document often follow a similar writing style, so data training set can be assumed to have the same feature distribution as the entire database. The accuracy of the model is about 95% by 3-cross validation in the training data.

7.3 Module generator

This module takes three responsibilities.

- Generating class for structure definition.
- Generating interface for DLL proxy
- Generating API Stub.

The approach of code generation is loading specifications from the extractor model into objects, and filling specifications into a dynamic template. This module is implemented in Java and Velocity script language. Velocity [4] is a template engine, which allow embedded script into a template to control the code generation process.

For a very simple example with Beep function (generates simple tones on the speaker):

```
BOOL WINAPI Beep(
  _In_ DWORD dwFreq,
  _In_ DWORD dwDuration
);
```

Its template is defined as below.

```
// Template in velocity script language
public class  $func.fname extends API {
  public $func.fname () {
    super();
    NUM_OF_PARMS = $func.fargs.size();
  }

  @Override
  public void execute() {
     long t1 = this.params.get(0);
     long t2 = this.params.get(1);

     $func.fargs1.type $func.fargs1.name = t1;
     $func.fargs2.type $func.fargs2.name = t2;

    int ret = ($func.flib).$func.fname($func.fargs1.name,
            $func.fargs2.name);
    $register.mov("eax", new LongValue(ret));
  }
}
```

In above template, $func stands for object API specification (e.g., $func.fargs.size() stands for the number of paramters, $func.flib stands for the library name, $func.fargs1.name stands for the first parameter name). The generated API stub is show as below.

[4]The Apache Velocity Project
http://velocity.apache.org/

```
public class Beep extends Kernel32API {
  public Beep () {
    super();
    NUM_OF_PARMS = 2;
  }

  @Override
  public void execute() {
    long t0 = this.params.get(0);
    long t1 = this.params.get(1);

    int dwFreq = t0;
    int dwDuration = t1;

    int ret = Kernel32DLL.INSTANCE.Beep(dwFreq, dwDuration);
    register.mov("eax", new LongValue(ret));
  }
}
```

7.4 Experiments

We extend the current version of BE-PUM developed at HCMUT with the generated API stub. The extended BE-PUM can analyze the malwares that are unsupported by the current BE-PUM.

Generated APIs stub

Based on about 1800 APIs description, our system generates:

- About 450 type definition classes
- 25 interface proxies
- About 1300 APIs stub.

However, our automatic stub generation fails on about 500 APIs, due to the presence of function pointers, void pointers and unknown parameters. In principle, function and void pointer can be resolved by copying the whole enviroment in BE-PUM to the working area of Windows API. However, this is heavy both in implementation and efficiency. Therefore, the current implementation choice is to copy in an on-demand way to reduce the size of copied data. For unknown parameters, we need to refine both module collector and NLP on API descriptions.

Experiments in unsupported malwares

The experiments are conducted with the unsupported malwares and the extensive version of BE-PUM. The unsupported malwares are real malware and unsupported by the current BE-PUM. In the extensive version of BE-PUM, we add new generated APIs stub and replaced current APIs stub. The result shows that the extensive BE-PUM can discover more nodes and edges in CFG of the unsupported malwares than the current version.

Windows XP SP3 32 bits and 2GB RAM are used for the experiments. Table 7.1 below compares the number of nodes, edges, running time between the extensive and current BE-PUM. The current BE-PUM may fail due to unsupported APIs stub, unknown x86 instructions or timeout. The last column shows reasons why the explosion of nodes occur, because the unsupported APIs are supported in the extensive version.

Virus	Extensive BE-PUM			Current BE-PUM			New supported APIs
	Time	Nodes	Edges	Time	Nodes	Edges	
0060e428f79cbe0408c1e4 6c8654ae10836d6d8883e dc9279d9d32e48413e574	45165ms	6869	7086	8372ms	70	69	GetSystemTimeAsFileTime MapVirtualKey
00787b7f34e8c30773903 6f03635f53d584f52ca632 b3598323405e6c5dd0984	116187ms	6617	7142	5698ms	64	63	RegisterWindowMessage
00b917d7a316390c1b995 e334628dcecdcfd6ae5f90 85994874fc331f5e3811f	46276ms	6996	7333	9624ms	69	68	CreatePen
00bcd33e9fb5ef7b9f2e54 b831d8166f46fbebf5b943 d827aa6dd59dec1358d7	245003ms	3315	3533	207528ms	186	209	HeapSetInformation
04542a21267a5fe668e42 af9b39d492e90889922fc 20453b7abfb9ffe128497b	5578ms	337	357	2113ms	34	34	VirtualLock

Table 7.1: Comparison of CFG construction

Experiments in supported malwares

The experiments are conducted with the supported malwares in both the extensive and current version of BE-PUM. The supported malwares are real malware and supported by the current BE-PUM. The purpose of the experiments is to verify whether the generated APIs stub give the same result as the manual APIs stub.

Chapter 8

Summary and Future Work

8.1 Conclusion and Current Limitation

Void pointer: In C, a void pointer is a generic pointer, and a void pointer variable can store the address of any kind of a variable. The compiler does not know what kind of objects a void pointer really points to. To do pointer arithmetic, what kind of objects pointed by pointer must be determined beforehand.

For example: general quick sort function is defined as below [1]

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

"base" is the address of an array, "nmemb" is the number of elements in the array, "size" is the size of each element, and "compar" is a pointer to a function that compares two elements. The qsort function (quick sort function) is considered as below.

```
int cmpfunc (const void * a, const void * b) {
   return ( *(int*)a - *(int*)b );
}
int main() {
  int values[] = { 88, 56, 100, 2, 25 };
  qsort(values, 5, sizeof(int), cmpfunc);
  return 0;
}
```

The kind of objects pointed by the void pointer in the above example is an array of integers. This occur in the same way with Windows API containing a void pointer parameter. From a binary analysis, we do not know what kind of object is pointed by void pointer parameter. Therefore, we do not know how many cells of simulation memory need to be

[1] C library function qsort()
http://www.tutorialspoint.com/c_standard_library/c_function_qsort.htm

copied to instantiate parameter objects for invoking API via JNA. For some APIs, even if a parameter is a void pointer, what kind of objects pointed to can be understood from its description.

Function pointer: Points to some code exists in a program. As in the above qsort example, "compar" is a pointer to a function that compares two elements. However, from binary analysis we do not know anything about the content of function pointed. As the same situation with the void pointer, we do not know how many cells of simulation's memory need to be copied to instantiate parameter objects for invoking API via JNA.

Due to such difficulty, Windows API contains a void pointer or a function pointer (i.e., callback function) are not generated at the moment.

The number of APIs: There are more than 4000 published Windows API document, however we collect only about 1800 APIs. Besides, the collected descriptions for structure are not complete.

8.2 Related Works

Automatically transforming from natural language description to programs attracts many attentions. Many approaches based on NLP to parse the semantic from natural language to a specific programming language, such as commands to robots [17], legal sentences to logic forms [18] and description to if-then-else code [19]. Another direction is programming by based examples, program is generated based on specifications extracted from given examples [20], and one of applications in real world is FlashFill function in Microsoft Excel 2013.

In our research, API description has specific characteristic, such as technique words and the mixture of natural language and C programming code. Furthermore, API stub generation does not require a full understanding. Therefore, we approach APIs stub generation by extracting API specifications from the its description, then fill specifications into dynamic templates. Because of special properties, the approach applies simple NLP techniques, but can achieve a good efficiency.

8.3 Future Work

To verify generated API stub, the conformance testing technique will be applied.

- Whether API specification extracted from API description is correct or not? Testing technique can be used to dis-ambiguous the results from NLP process. For example, to check whether memory length parameter is correct, we can generated programs, and execute them to see the result. However, we need assumptions about valid inputs of all parameters.

- Whether generated API stub is correct. We can compare the result after API call between BE-PUM and a debugger (e.g., OllyDbg and Intel-Pin). However, there are many difficulties, such as valid input condition and dependency between API

specifications. Besides, we have to obtain the value from a debugger at the time after calling API as an engineering difficulty.

Bibliography

[1] P. Szor, *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[2] F. Song and T. Touili, "Pommade: Pushdown model-checking for malware detection," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, (New York, NY, USA), pp. 607–610, ACM, 2013.

[3] A. Holzer, J. Kinder, and H. Veith, "Using verification technology to specify and detect malware," in *Proceedings of the 11th International Conference on Computer Aided Systems Theory*, EUROCAST'07, (Berlin, Heidelberg), pp. 497–504, Springer-Verlag, 2007.

[4] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," in *Proceedings of the Second International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'05, (Berlin, Heidelberg), pp. 174–187, Springer-Verlag, 2005.

[5] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Proactive detection of computer worms using model checking," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, pp. 424–438, Oct 2010.

[6] S. Nanda, W. Li, L.-C. Lam, and T.-c. Chiueh, "Bird: Binary interpretation using runtime disassembly," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, (Washington, DC, USA), pp. 358–370, IEEE Computer Society, 2006.

[7] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "Codesurfer/x86—a platform for analyzing x86 executables," in *Proceedings of the 14th International Conference on Compiler Construction*, CC'05, (Berlin, Heidelberg), pp. 250–254, Springer-Verlag, 2005.

[8] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The bincoa framework for binary code analysis," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, (Berlin, Heidelberg), pp. 165–170, Springer-Verlag, 2011.

[9] S. Bardin, P. Baufreton, N. Cornuet, P. Herrmann, and S. Labb, "Binary-level testing of embedded programs," in *2013 13th International Conference on Quality Software*, pp. 11–20, July 2013.

[10] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proceedings of the 2007 ACM Workshop on Recurring Malcode*, WORM '07, (New York, NY, USA), pp. 46–53, ACM, 2007.

[11] T. Izumida, K. Futatsugi, and A. Mori, "A generic binary analysis method for malware," in *Proceedings of the 5th International Conference on Advances in Information and Computer Security*, IWSEC'10, (Berlin, Heidelberg), pp. 199–216, Springer-Verlag, 2010.

[12] N. M. Hai, M. Ogawa, and Q. T. Tho, *Obfuscation Code Localization Based on CFG Generation of Malware*, pp. 229–247. Cham: Springer International Publishing, 2016.

[13] T. M. Mitchell, *Machine Learning*. New York, NY, USA: McGraw-Hill, Inc., 1 ed., 1997.

[14] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*. The MIT Press, 2012.

[15] P. J. Parag Kulkarni, *Artificial Intelligence: Building Intelligent Systems*. PHI Learning Pvt. Ltd., 1 ed., 2015.

[16] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[17] R. J. Kate, Y. W. Wong, and R. J. Mooney, "Learning to transform natural to formal languages," in *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3*, AAAI'05, pp. 1062–1068, AAAI Press, 2005.

[18] M. Nakamura, S. Nobuoka, and A. Shimazu, "Towards translation of legal sentences into logical forms," in *Proceedings of the 2007 Conference on New Frontiers in Artificial Intelligence*, JSAI'07, (Berlin, Heidelberg), pp. 349–362, Springer-Verlag, 2008.

[19] C. Quirk, R. Mooney, and M. Galley, "Language to code: Learning semantic parsers for if-this-then-that recipes," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL-15)*, (Beijing, China), pp. 878–888, July 2015.

[20] S. Gulwani, "Programming by examples (and its applications in data wrangling)," in *Verification and Synthesis of Correct and Secure Systems* (J. Esparza, O. Grumberg, and S. Sickert, eds.), IOS Press, 2016.