DETECTING MALICIOUS BEHAVIOR IN BINARY PROGRAMS USING DYNAMIC SYMBOLIC EXECUTION AND API CALL SEQUENCES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS OF
THE MIDDLE EAST TECHNICAL UNIVERSITY
BY

FATİH TAMER TATAR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

JUNE 2021

Approval of the thesis:

DETECTING MALICIOUS BEHAVIOR IN BINARY PROGRAMS USING DYNAMIC SYMBOLIC EXECUTION AND API CALL SEQUENCES

Submitted by **FATİH TAMER TATAR** in partial fulfillment of the requirements for the degree of **Master of Science in Information Systems Department, Middle East Technical University** by,

Prof. Dr. Deniz Zeyrek Bozşahin
Dean, **Graduate School of Informatics** _____

Prof. Dr. Sevgi Özkan Yıldırım
Head of Department, **Information Systems** _____

Assoc. Prof. Dr. Aysu Betin Can
Supervisor, **Information Systems Dept., METU** _____

Examining Committee Members:

Assoc. Prof. Dr. P. Erhan Eren
Information Systems Dept., METU _____

Assoc. Prof. Dr. Aysu Betin Can
Information Systems Dept., METU _____

Assoc. Prof. Dr. Banu Günel Kılıç
Information Systems Dept., METU _____

Assoc. Prof. Dr. Altan Koçyiğit
Information Systems Dept., METU _____

Assoc. Prof. Dr. Ayça Kolukısa Tarhan
Computer Engineering Dept., Hacettepe University _____

Date: **18.06.2021**

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Fatih Tamer TATAR

Signature : _____

ABSTRACT

DETECTING MALICIOUS BEHAVIOR IN BINARY PROGRAMS USING

DYNAMIC SYMBOLIC EXECUTION

AND

API CALL SEQUENCES

Tatar, Fatih Tamer

MSc., Department of Information Systems

Supervisor: Assoc. Prof. Dr. Aysu Betin Can

June 2021, 53 pages

Program analysis becomes an important part of malware detection as malware become stealthier and more complex. For example, modern malware may detect whether they are under analysis and they may use certain triggers such as time to avoid detection. However, current detection techniques turn out to be insufficient as they have limitations to detect new, obfuscated, and intelligent malware. In this thesis, we propose a behavior based malware detection methodology using API call sequence analysis. In our methodology, we combine dynamic symbolic execution and API function models to extract call sequences of a given binary program and decide whether it has a malicious sequence. In our experiments, we showed that our methodology is capable of detecting malware hiding behind evasion techniques and our methodology is applicable to a real-world problem.

Keywords: Call Sequence Analysis, Dynamic Symbolic Execution, Function Modeling, Malware Analysis

ÖZ

DİNAMİK SEMBOLİK UYGULAMA VE
API ÇAĞRI SIRALAMALARI KULLANARAK
İKİLİ PROGRAMLARDA ZARARLI DAVRANIŞ TESPİTİ

Tatar, Fatih Tamer

Yüksek Lisans, Bilişim Sistemleri Bölümü

Tez Yöneticisi: Doç. Dr. Aysu Betin Can

Haziran 2021, 53 sayfa

Kötü amaçlı yazılımlar daha gizli ve daha karmaşık hale geldikçe program analizi, kötü amaçlı yazılım tespitinde önemli bir parça haline gelmiştir. Örneğin, modern kötü amaçlı yazılımlar programın analiz altında olup olmadıklarını tespit edebilir ve tespit edilmekten kaçınmak için zaman gibi belirli tetikleyicileri kullanabilirler. Ancak mevcut tespit yöntemleri yeni, karıştırılmış ve akıllı kötü yazılımları tespit etmekteki sınırlamaları sebebiyle yetersiz kalmaktadır. Bu tezde, API çağrı dizisi analizini kullanan davranış tabanlı bir kötü yazılım tespit methodolojisi önerilmektedir. Metodolojimizde, verilen programın çağrı dizilerini çıkarmak ve kötü amaçlı bir diziye sahip olup olmadığına karar vermek için dinamik sembolik yürütme ve API fonksiyon modelleri birleştirilmiştir. Deneylerimiz ise metodolojimizin, kaçınma tekniklerinin arkasında saklanan kötü amaçlı yazılımları tespit edebildiğini ve gerçek dünyadaki bir soruna uygulanabilir olduğunu göstermektedir.

Anahtar Kelimeler: Çağrı Sırası Analizi, Dinamik Sembolik Yürütme, Fonksiyon Modelleme, Kötü Amaçlı Yazılım Analizi

To my mother, Ayşe,

To my father, Mehmet,

To my sisters, Çiğdem and Mehtap.

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisor Assoc. Prof. Dr. Aysu Betin Can for her support, guidance, criticism, inspiration and insight throughout the research.

Besides my supervisor, I would like to acknowledge my team leader, Barış İyidir, and my manager, Miren Izaskun Gallastegi Dorken, from ASELSAN for letting me complete my graduate studies.

I would also like to thank my valuable friends Elif, Gizem, Didem and Şafak for their endless support and motivation to finish this thesis.

Lastly, I would like to express my gratefulness to my sisters Mehtap Tatar and Çiğdem Yüksel for being perfect role models for my entire education and my parents Ayşe Tatar and Mehmet Tatar for always being supportive throughout my decisions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
CPU	Central Processing Unit
DLL	Dynamic Link Library
MSDN	Microsoft Software Developer Network

CHAPTER 1

INTRODUCTION

It has been an issue to detect malicious software for decades. Researchers use various techniques to discover malware such as signature, behavior, and deep learning based techniques. With the development of complex and obfuscated malware, traditional malware detection techniques such as signature-based detection become inadequate to unknown malware [1] and there is a demand for more scientific studies to cover the shortcomings of existing methods. [2]

As malware analysis techniques improve, malware becomes stealthier and more intelligent. New malware may not show their malicious behavior immediately especially if they are aware of being analyzed. For example, the dropper component of WannaCry ransomware [3] tries to access an invalid resource, an unregistered domain name, to detect if it is running in a sandbox environment. In addition, new malware may postpone their malicious activity until they are triggered by a resource such as a keyboard. For instance, MyDoom waits until February 1 and 3, 2004 to perform its DDOS attack [4]. Therefore, malware detection becomes a challenging task. It is also very important to understand the malware behavior on the infected system. If a malware analyst is late to take precautions, undetected malware behavior may help the malware to persist in the system under different forms or even worse, its spread may not be prevented at all and everything goes off the spin. Consequently, it is not enough to detect the malware only. Analysts should also need to understand its effects on the system. However, the latest malware detection methodologies fail to propose a general solution for these problems. For example, signature-based detection is adapted by security companies for its quickness, but they fail to detect unknown [1] and obfuscated malware [5]. Deep learning based detection methods perform up to 95-99% success rates [6]. However, they suffer from reliability [7]. Behavior based malware detection methods run the malware in a contained environment and detect malware even its code changes as long as its core functionality stays the same. Nevertheless, they fail to detect intelligent malware which is capable of sensing analysis environment [8]. In our work, we

approach the problem from the behavior-based malware detection perspective and combine API analysis approach using the power of dynamic symbolic execution.

In this thesis, we developed a methodology for analyzing suspicious binary programs using dynamic symbolic execution to observe API call sequences. In our work, we aim to detect given malicious behavior and provide a methodology to analyze malware behavior using API call sequence analysis for the Windows platform. To show the feasibility of our work, we also developed a toolset for the methodology. First, our toolset extracts the Windows API functions used by the binary program and creates models for the extracted functions. Then, it replaces actual functions with the function models and executes the binary program using dynamic symbolic execution. In this way, our function models create traceable outputs for the API calls and our toolset extracts the call sequences of all possible program branches even the branches are hidden behind certain trigger conditions such as sandbox evasion and time. Next, our toolset compares the extracted call sequences with known malicious API call sequences provided by the user. If our toolset finds a matching sequence, it displays a warning and presents evidences of malicious behavior. As our methodology utilizes dynamic symbolic execution on a binary program, it is resilient to obfuscation methods such as packaging and capable of detecting new malware once the user provides a malicious API call sequence. Also, our toolset supports extensibility as users may modify its default behavior to meet their future needs and change malicious API call sequence input to employ the future developments in the literature.

In order to show the effectiveness of our methodology, we conducted experiments on a synthetic and a real-world malware, WannaCry. During the experiments, our toolset generated more than 1200 lines of C++ code to model 75 Windows API functions automatically. The synthetic malware experiment showed our capability of detecting a malicious behavior, DLL injection, even it is hidden behind time discovery and sandbox evasion techniques. On the other hand, WannaCry ransomware experiment showed the applicability of our methodology to a real-world problem. In the experiment, our methodology discovered behaviors such as registry key creation, file hiding, file access modifications and function imports for encryption purposes.

Overall, we make the following contributions:

- We present an extensible toolset for analyzing binary programs that supports future developments in the malicious call sequence analysis area.
- We propose a technique to observe API call sequences using function models.
- We introduce an approach that combines function models and dynamic symbolic execution to analyze malware without being affected by obfuscation techniques such as time discovery and sandbox evasion.

- We implement a method to avoid state space explosion problem of symbolic execution by changing return values of API models to symbolic or concrete without recompiling the models.
- We show evidences after the analysis to support decision-making and increase user benefit.

The rest of the thesis is organized as follows. Chapter 2 introduces the related work in terms of background information and the literature review. Chapter 3 presents the proposed methodology in detail. Chapter 4 describes the experimental work showing the feasibility of our methodology. Chapter 5 concludes our study with the limitations and the future work.

CHAPTER 2

RELATED WORK

2.1 Background Information

In this section, we introduce general concepts and terminology in the thesis to provide an overview of the topic.

Malware. Cyber attackers design malicious software programs, also known as malware, to steal personal data, gain financial benefits, and damage devices. Malware can be labeled into different categories such as trojan horses, worms, polymorphic viruses, and ransomware[5]. Attackers use trojan horses to hide the malware inside other programs that appear to be innocent. Worms are malware that spread to other devices by copying themselves. They may infect a computer network without any manual intervention. Polymorphic viruses are one of the hardest malware to analyze. They evade detection systems by changing themselves in runtime. For example, they may modify their code without changing the main functionality, decrypt or unpack previously hidden malicious code segments. Attackers also use ransomware to gain financial benefits by encrypting personal data and demanding ransom for decryption.

Dynamic Symbolic Execution. Symbolic execution is a program analysis technique that analyzes the programs by traversing all possible branches and generating constraints for them. The pioneers of the technique such as DART[9], CUTE[10], KLEE[11], and SAGE[12] use symbolic execution to find program bugs by using constraint solvers. In 2005, CUTE improved the symbolic execution and introduced the concept of concolic (**conc**rete symb**olic**) execution, also known as dynamic symbolic execution. In their work, they combined concrete and symbolic execution to create test inputs for the discovery of all possible execution branches.

Symbolic execution performs as follows: First, a symbolic execution engine replaces program inputs with symbolic variables that can hold any value. Symbolic variables are analogous to the mathematical unknown variables such as X, Y, Z. Then, when program execution reaches a branch using a symbolic variable, the symbolic execution engine executes both branches simultaneously and creates a set of constraints called path condition. Path conditions are mathematical formulas that represent a valid range of values for symbolic variables to satisfy the current branch condition. When the path reaches a termination point or a bug, the symbolic execution engine uses a constraint solver to evaluate the satisfiability of the path condition. If the condition is mathematically solvable, the solver returns a concrete value for the symbolic variable and this value may be used as a test condition for the path [11]. However, if the condition is unsolvable, symbolic execution stops.

On the other hand, dynamic symbolic execution starts the program with random concrete values as inputs. Then, during the execution, it keeps track of both concrete values and symbolic constraints. When execution reaches a termination point, the engine returns to the branch point and negates the constraint in order to decide if there is an input that satisfies the other branch. If such input exists, the engine uses the newly found concrete value to continue execution. However, if the constraint is too complex to solve, it simplifies the constraint using concrete values and the constraint solver generates such an assignment. Then, the engine runs the program with these concrete inputs.

One of the biggest drawbacks of symbolic execution is the path explosion problem. As symbolic execution engine runs many branches simultaneously, it starts to suffer from high memory consumption. In order to solve this problem, symbolic execution engines utilize different techniques such as prioritized path searching [13] and constraint optimization [12]. In addition, there are symbolic execution engines [14] [15] that directly run on binary programs where it is very useful when there is no access to the code of the program such as malware.

API Modeling. Programs running in the user space need to call Application Programming Interface (API) functions to use the services provided by the operating system's kernel. So, in order to understand the main behavior of a program, the sequence of its API calls can be analyzed. Similarly, malware analysis techniques [16] [17] also use API call sequence information to analyze malware behavior. In our work, in order to collect the API call sequence of malware in run-time, we re-write the Windows API functions in a way that they do not perform their actual tasks. In other words, we create models of the Windows API functions to understand the behavior of malware while it is running. Instead of performing real API activities, our models create logs upon execution.

Dynamic Linking. Windows provide its API in the form of Dynamic Link Libraries (DLLs). DLL files enable the share of functions and resources among different programs by allowing programs to use a single DLL file in memory at the same time. So, DLL

files save memory and disk space. In contrast to static linking where the content of a static library is duplicated into the programs, dynamic linking creates only the information needed by Windows at runtime to find the DLL file containing data or function [18]. In our work, we use a technique, DLL injection, to overwrite the dynamic linking behavior of Windows so that we insert our API models into malware to understand its behavior.

2.2 Literature Review

As new and complex malware emerge, detection methods also evolve rapidly. We focus on three main malware detection techniques in the literature, signature-based, deep learning based and behavior based. Then, we present other approaches in literature such as API analysis and symbolic execution.

In the early studies, the signature-based detection method is widely accepted by antivirus vendors as it provides a quick and effective way of detecting known malware [2]. Researchers extract malware signatures in different ways, such as integrity checking [19], string scanning [20], top and tail scanning [21], and entry point scanning [21]. However, these techniques are not capable of detecting new unknown malware [1] since there is no signature match for the new malware in the signature database. Also, signature-based detection techniques suffer from malware using obfuscation techniques [22], such as encryption, packaging, and polymorphism, and require continuous updates of signature databases which require maintenance cost. Though there are studies [23] [24] [25] making improvements to overcome obfuscation methods, their success is still limited to polymorphic malware. In our study, we are not limited to detect known malware. Although our methodology is still limited to the malicious call sequence input, our methodology provides an extensible approach for the detection of new unknown malware using API call sequences. Once the user gives the malicious call sequence, our methodology can detect an unknown malware performing such sequence. Also, our configurable toolset provides an analysis environment where the users may incorporate their expertise in the area to compose new sequences.

Studies using deep learning for malware detection mainly focus on four techniques namely multilayer perceptrons (MLP) [26] [27] [28], convolutional neural networks (CNNs) [29] [30] [31], recurrent neural networks (RNNs) [32] and Hybrid Models [33]. Although they show high malware detection rates up to 95-99% [6], they suffer from effectiveness and reliability [7]. Also, they are not resistant to malware using perturbation [34] [35] and evasion [36] techniques. Furthermore, 70% of the deep learning studies detecting malware focus on Android devices [6]. Our methodology fills the gap of reliable malware detection methodology for the Windows platform.

Behavior based malware detection techniques determine whether a program is benign or malware using monitoring tools and sandboxes [37] such as Norman Sandbox [38]. Also, they detect malware even if malware code changes as long as the behavior stays the same [37]. However, the main disadvantage of behavior based malware detection is that malware may detect the analysis environment and it may avoid showing its malicious behavior under analysis [39] [8]. In our approach, we used a behavior based malware detection technique and solved this problem by combining dynamic symbolic execution and API function models that return concrete or symbolic values. So, our models can distort the malware's perception about its environment and provide information in user control.

In order to detect malware, researchers use the similarity between the API calls of the new and the known malware. In terms of their approaches, their API call analysis methods divide into two categories: static [40] [41] [42] and dynamic [43] [44] [45]. Researchers using dynamic API call analysis analyze the malware in runtime and they achieve better results analyzing obfuscated malware with respect to static API call analysis researchers. [40] [43] analyze malware to extract the frequency of repeatedly used API functions and their total events. Moreover, [44] [45] use API calls to extract static signatures but they fail to detect polymorphic and unknown malware. Also, [46] states that studies using API call sequences suffer from the fuzzy API calls that attackers intentionally insert, delete, replace existing ones without affecting the overall functionality. [16] used dynamic analysis to extract API calls of more than 23000 malware and apply DNA sequencing algorithm to find critical API call sequence patterns. However, their approach fails to detect malicious behavior hidden behind the logic bombs and evasion techniques. In our methodology, we fill this gap by detecting hidden malware behavior by utilizing API function models in a dynamic symbolic execution. Also, we provide a configurable tool set that users may add the latest API call sequence information in the literature so that our methodology provides an up-to-date solution. Furthermore, malicious API call sequence information lets our methodology distinguish previously unknown malware.

Since the first introduction of symbolic execution [47], it is used in different areas such as test case generation [9] [10] [11] [12] [48], bug discovery [11] [13] [14] [49] [50] and malware analysis [51] [52] [53] [4]. Although [51] successfully detects a remote access Trojan (RAT) using the symbolic execution framework ANGR [15], their work is limited to RAT families. On the other hand, [4] and [53] conduct similar work. They approach malware detection with a broader aspect and analyze the existence of trigger sources and their corresponding conditions, such as system time, system event keyboard inputs, and system calls. However, their approach is still limited to trigger sources. Our methodology provides an extensible API call sequence analysis where users may modify the control mechanism, call sequence, and API function models to meet their future needs.

Since our methodology utilizes malicious API call sequences, it may seem to be a signature-based malware detection technique. However, signature-based malware detection techniques use strings, byte sequences, entry points, and integrity checks of the binary program as a signature. Moreover, our methodology focus on the run-time behavior of the binary program to extract invocations of Windows API functions and runs the binary program in a symbolic execution environment. So, our methodology can be positioned as a behavior-based malware detection technique. Table 1 shows the summary of the techniques used for malware detection.

Table 1: Summary of the Techniques Used for Malware Detection

Technique	Pros	Cons
Signature Based	Quick and effective [2]	• Fails to detect unknown [1] and obfuscated malware [5] • Requires maintenance cost for the continuous updates of the signature database
Deep Learning Based	High (95-99%) success rate [6]	• Suffers from effectiveness and reliability [7] • Not resistant malware using perturbation [34] [35] and evasion [36] techniques
Behavior Based	Detects malware even if the malware code changes as long as malware behavior stays the same [37]	• Fails to detect intelligent malware which is capable of sensing analysis environment [8]
API Call Analysis	Disclose the attributes of the malware in the same class [16]	• Suffers from fuzzy API calls [46] • Cannot detect logic bombs and evasion techniques
Symbolic Execution	Capable of detecting trigger sources, time bombs, and evasion techniques	• Suffers from state space explosion problem • Takes more time with respect to signature based detection techniques

CHAPTER 3

METHODOLOGY

In this thesis, we developed a methodology to analyze suspicious binary programs written for the Windows platform. In addition, we designed a toolset to show the feasibility of our methodology. We aim to detect whether a given program may generate a malicious sequence of function calls using dynamic symbolic execution. Our tool takes suspicious and malicious function call sequences and examines the binary program whether such sequence is possible. The tool generates evidence showing what data lead the program to produce such a malicious execution sequence. Our approach consists of 8 steps and we provide the details in the following sections of this chapter. We show an illustrative summary of our work in Figure 1.

In order to make the function calls traceable and facilitate the reachability of different execution paths of the binary program, we use a decompiler and extract the functions called from the Windows API. Then, our model generator module creates models for these functions where they emit execution information such as their function names and their arguments. Next, we combine function models and create a DLL file. After that, an injector application injects the DLL file into the binary program. So, the program calls the modeled functions instead of calling actual Windows API functions.

To initiate every API function call combination in the program, we need to traverse all the possible execution paths. Therefore; we used a symbolic execution framework called S2E to run the program symbolically. It invokes all possible function call combinations along the paths of the binary program. At this point, function models facilitate the symbolic execution as they cost less than actual API functions. Before the execution, we also set the execution environment and determine a time-bound for the analysis; so that our analysis does not suffer from the state-space explosion and endless consumption of the resources. While the framework is running every possible path of the program, it calls our modeled functions in the execution order. Then, our analysis parser module

collects execution information, such as function names and their arguments, and processes it to generate the program's function call sequence.

After that, the log analyzer compares the call sequence with the malicious sequences given at the beginning of the analysis. As a result, it warns the user if it finds a correspondence between these sequences and shows the evidences of its findings.

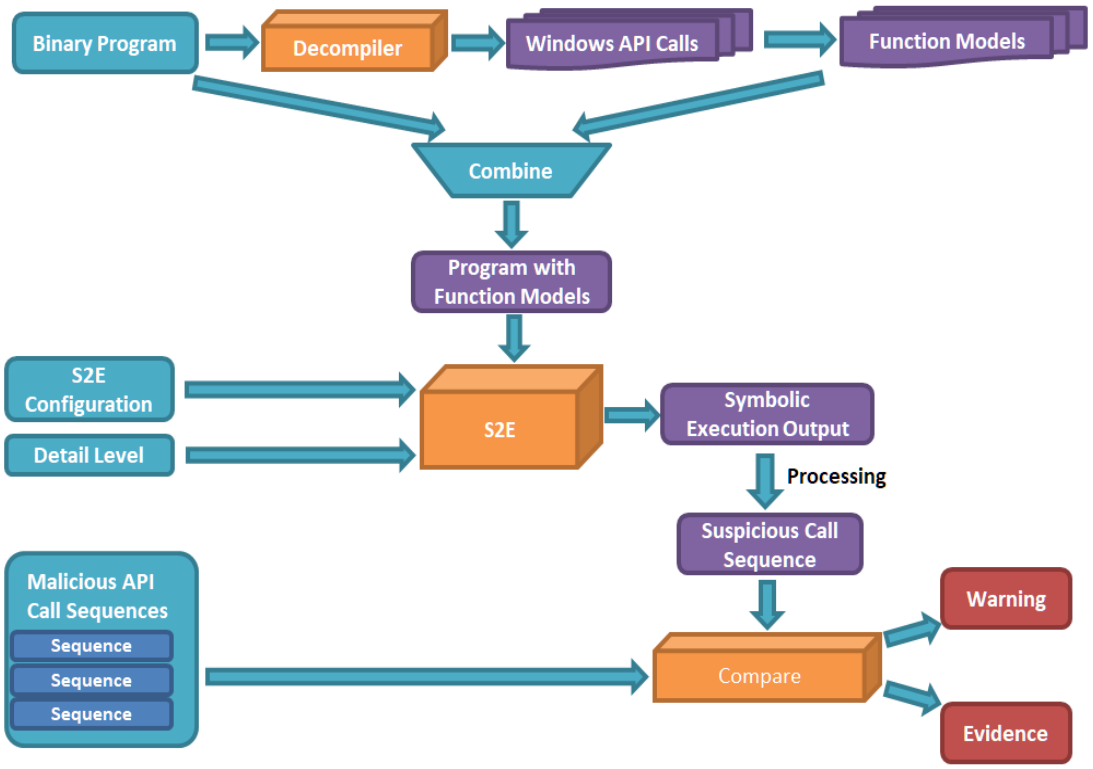

Figure 1 : Summary of Proposed Methodology

3.1 Input Gathering

Our system takes four inputs. Namely, a binary program, a malicious function call sequence, a detail level, and a configuration. The program is a 32-bit Windows executable that we suspect its behavior of hiding malicious activities.

The second input is the malicious function call sequence specification written in a file. Users either use their expert knowledge or the latest developments in the API call sequence analysis literature to create the content of this file. Each line of the file

corresponds to one call sequence and each of them starts with a sequence name. A sample call sequence is shown in Figure 2. In the figure, there are two sequences namely IATHooking and DLLInjection [16]. After the sequence name, lines continue with comma symbols and function names. Precedence between the functions determines the order in the sequence. For example, the second line of the figure means the following: DLLInjection sequence starts with OpenProcess function followed by VirtualAllocEx function. Then, the WriteProcessMemory function comes and the sequence ends with the CreateRemoteThread function. A pipe symbol "|" in a sequence indicates function variation. For instance, the LoadLibrary function in the IATHooking sequence can be followed by either strcmp, strncmp, _stricmp or strnicmp function.
The third input is the detail level. It determines the analysis detail by deciding the return values of the modeled functions to be concrete or symbolic values. Symbolic values will make the symbolic engine to try different execution paths while concrete values make the engine choose one of the possible executions. Concrete return values make the analysis less detailed. As the number of the symbolic return values increases, the symbolic execution framework becomes more likely to discover new paths; hence, it discovers hidden sequences within the binary program. Our system takes the detail level as a command line input in the form of a free text. We use this free text to specify the function names that we want to return symbolic values or write 'all' to represent all function names. Later, function models interpret this free text to decide their return value.

```
1 IATHooking,LoadLibrary,strcmp|strncmp|_stricmp|strnicmp,VirtualProject
2 DLLInjection,OpenProcess,VirtualAllocEx,WriteProcessMemory,CreateRemoteThread
```

Figure 2 : Sample Malicious Function Call Sequence

3.2 Extracting Dynamically Linked Functions

In order to track the function call sequence of the binary program; first, we need to determine the function calls made by the program. To demonstrate the feasibility of our approach, we narrow the scope of our work to Windows API functions. Since these functions are linked dynamically, we further narrowed our analysis to dynamically linked functions.

We used a decompiler, Retargetable Decompiler (RetDec) [54], to extract all dynamically linked Windows API functions in the binary program. Although this tool is known for its decompiling capabilities, we used it to analyze dynamically linked functions because it supports cross-platform analysis and provides an insight for the statically linked functions. Despite our analysis does not cover statically linked functions, we aim to add this feature to analyze the whole program in the future. An alternative to decompiler would be Microsoft's dumpbin.exe; however, it only runs on

Windows machines and does not capture the statically linked functions. Consequently, we decided to use a tool that both meets our needs and supports our vision for the future. A sample output of the decompiler is in Figure 3.

```
// --------------- Statically Linked Functions ---------------

// int32_t _3f__3f_2_40_YAPAXI_40_Z(int32_t al);
// int32_t _3f__3f__G__non_rtti_object_40_std_40__40_UAEPAXI_40_Z(int32_t al);
// int32_t _3f__3f__G_Ref_count_base_40_std_40__40_UAEPAXI_40_Z(int32_t al);
// int32_t _40___security_check_cookie_40_4(void);

// --------------- Dynamically Linked Functions ---------------

// int32_t ?_Xbad_alloc@std@@YAXXZ(void);
// int32_t _3f__Xinvalid_argument_40_std_40__40_YAXPBD_40_Z(char * al);
// int32_t _3f__Xlength_error_40_std_40__40_YAXPBD_40_Z(char * al);
// int32_t _3f__Xout_of_range_40_std_40__40_YAXPBD_40_Z(char * al);
// _ACRTIMP_ALT FILE * __cdecl __acrt_iob_func(unsigned);
// int __cdecl __stdio_common_vfprintf(_In_ unsigned __int64 Options,
    _Inout_ FILE * Stream,
    _In_z_ char const * Format,
    _In_opt_ _locale_t Locale, va_list ArgList);
// int * __cdecl _errno(void);
// void __cdecl _invalid_parameter_noinfo_noreturn(void);
// BOOL CloseHandle(_In_ HANDLE hObject);
// HANDLE CreateRemoteThread(_In_ HANDLE hProcess,
    _In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_ SIZE_T dwStackSize, _In_ LPTHREAD_START_ROUTINE lpStartAddress,
    _In_opt_ LPVOID lpParameter,
    _In_ DWORD dwCreationFlags,
    _Out_opt_ LPDWORD lpThreadId);
// void exit(int status);
// void free(void * ptr);
// int getchar(void);
// HMODULE GetModuleHandleW(_In_opt_ LPCWSTR lpModuleName);
// FARPROC GetProcAddress(_In_ HMODULE hModule, _In_ LPCSTR lpProcName);
// void * memcpy(void * restrict dest, const void * restrict src, size_t n);
// void * memmove(void * dest, const void * src, size_t n);
// HANDLE OpenProcess(_In_ DWORD dwDesiredAccess,
    _In_ BOOL bInheritHandle, _In_ DWORD dwProcessId);
// LPVOID VirtualAllocEx(_In_ HANDLE hProcess, _In_opt_ LPVOID lpAddress,
    _In_ SIZE_T dwSize, _In_ DWORD flAllocationType, _In_ DWORD flProtect);
// long int wcstol(const wchar_t * restrict nptr,
    wchar_t ** restrict endptr, int base);
// BOOL WriteProcessMemory(_In_ HANDLE hProcess, _In_ LPVOID lpBaseAddress,
    LPCVOID lpBuffer, _In_ SIZE_T nSize,
    _Out_opt_ SIZE_T * lpNumberOfBytesWritten);
```

Figure 3 : Sample Decompiler Output

3.3 Automatic Function Model Generation

In this step, we process the decompiler's output and create models for the Windows API functions called by the binary program. To automate this process, we developed a model generator as a module in Python.

Since we decided to work with dynamically linked functions, we need to extract their declarations from the output file of the decompiler. So, the model generator first collects all of them into a file. Then for each function declaration, it creates method bodies. A method body in a function model does not perform the actual responsibility of that function nor calls any of the Windows functions. Instead, it emits the function name and values of its arguments and then returns either a symbolic or predetermined value depending on the analysis level. We give a sample function model in Figure 4.

```
1  static HANDLE WINAPI OpenThread_Model(DWORD dwDesiredAccess, BOOL bInheritHandle,
2                                        DWORD dwThreadId)
3  {
4      Message("OpenThread_Model\n");
5      Message("[OpenThread_Model|evidence] {[DesiredAccess: %d,\
6              Does it inherit handle?(0:false/1:true): %d,\
7              Thread Id: %d\n",dwDesiredAccess,bInheritHandle,dwThreadId);
8
9      HANDLE retVal = NULL;
10     if ((detail.compare("all") == 0) || (detail.find("OpenThread_Model") != -1))
11     {
12         S2EMakeSymbolic(&retVal, sizeof(retVal), "OpenThread_Model");
13     }
14     else
15     {
16         /* Concrete output */
17     }
18
19     return retVal;
20 }
```

Figure 4 : Sample Function Model

A model of a function, when replaced with the corresponding actual function, will provide an execution trace. A trace, in our methodology, is an execution log showing the functions called in the order they are made. When executing the program in question, a trace is built by using messages showing the name of the functions. For example, line 4 in Figure 4 shows a sample log message. Whenever the symbolic execution engine invokes the model function, it emits the function name.

In addition to execution trace, a function model provides information to support the result of the call sequence analysis. We call function evidence to this information. Each

model includes log messages, function evidences, to display their arguments. Upon invocation, models emit their argument names and values in a human-readable format. Since our generator does not support human readability, we manually edit the evidences to increase their understandability. To illustrate, lines 5, 6, and 7 in Figure 4 show sample evidence. Later, the log analyzer module displays evidences to support the analysis result. We explain the usage of evidences in section 3.7.

Moreover, the generated function bodies have return values coherent with the function declaration. Although there are infinitely many possibilities, we wanted to work with a constraint set of simple return values. As a result, we determined to use the values given in Table 2. These are not the final values that are supposed to fit all cases; instead, we use them to show the feasibility of our approach. Users of the system may alter these values to enhance this table according to the future need of the analysis.

Besides concrete return values, our model generator is also capable of producing function bodies returning symbolic values. In Figure 4, line 12 shows this capability. The symbolic values facilitate the path exploration in the symbolic execution. Even though the model generator is capable of returning all the values symbolically, we decided to use it with caution in order not to have a state-space explosion problem during symbolic execution. Nevertheless, our approach is not limited to our choice of implementation. It can be extended to meet different needs. For example, if a detailed analysis is required, other functions may also return symbolic values instead of concrete ones. Line 10 of Figure 4 shows a sample conditional statement for the management of return values. Modifying a file content extracted in Section 3.4, users may decide the return values of the models to either symbolic or concrete.

Table 2: Concrete Return Types and Their Values

Return Type	Value		Return Type	Value
BOOL	TRUE		int*	NULL
DWORD	1		int32_t	1
FARPROC	NULL		long	1
FILE*	NULL		LPVOID	NULL
HANDLE	NULL		SIZE_T	1
HMODULE	NULL		UINT	1
int	1		void*	NULL

3.4 Functions Replacement

In the previous step, the model generator created function models emitting their name, arguments, and argument values; as well as, returning concrete or symbolic values depending on the detail level. In this step, we used a technique called DLL injection to replace actual functions of the binary program with the function models so that function calls of the binary program become traceable.

As the technique suggests, we create a DLL file and put our function models inside. Then, we use an injector module to replace actual API functions with the function models. Next, the injector application runs the binary program in a suspended state and injects the DLL file. After the injection, it resumes the execution of the binary program.

3.4.1 DLL Creation

In order to put our function models inside a DLL file, we used EasyHook library [55] for its simplicity. First, we changed our model names so that they do not exactly match with the real API functions in order not to have any conflicts. Next, for every function model, we need to match the name of the function with its actual Windows API equivalent along with the corresponding library name. In order to find the library name, we write a library finder module in Python to automate this process. It visits MSDN pages using Selenium [56], a suite of tools for automating web browsers, and matches Windows API functions with their libraries. Then, we provide this match information to the EasyHook via its API. In the end, we get a DLL file ready to inject where we use injected function models' outputs to trace the call sequence of the binary program.

In this step, we add a feature inside the DLL file that facilitates the change of model behaviors. Although we use this feature to decide whether function models return concrete or symbolic values, it also provides a capability to switch between different concrete values. For this purpose, we used a file to configure models from the outside; so that, we can change their behavior by only editing a file content. As a result, we eliminate the heavy weight of recompiling all the models over and over again if we need a simple change in the model. We show the code snippet that we read external input in Figure 5.

```
/* Get detail level */
ifstream detailFile("detailLevel.txt"); //open

while (getline(detailFile, detail)) // read
{
    Message(detail.c_str()); // log
}

detailFile.close(); //close
```

Figure 5 : Detail Level Snippet

We focused on using this feature to change the detail level of our analysis. To do so, we used a file called detailLevel.txt containing data for the detail level. Later, the DLL file reads detailLevel.txt, and models either return symbolic or concrete values according to our choice of detail. Please also note that, while creating the models, we modeled our functions so that they support this feature. Figure 4 shows the detail level information inside a function model.

3.4.2 DLL Injector Application

In the previous step, we created a DLL file having instructions to replace the real Windows API functions with our models. In order to complete function hooking, we need to insert the DLL file into the binary program. As in the previous step, we used EasyHook's API to facilitate the DLL injection and we implemented an injector application that inserts the DLL file. Our injector starts the binary program in a suspended state so that the program waits without calling any functions. Later, our injector inserts the DLL file to replace actual functions with the model functions. Then, it wakes up the program and the program starts to run. In other words, our injector behaves like malware as it runs another program after changing its behavior. This behavior allowed us to insert our model functions.

In addition to the injection, our injector takes command-line arguments to decide whether the user asks for a detailed analysis or not. In other words, our injector module supports the feature that we use to change the model behavior without recompiling the model codes. After the injector takes the command line argument, it creates the detailLevel.txt containing the command line value. Then, the DLL file read this file to capture detail level.

3.5 Running Symbolic Execution

In order to trigger all the function calls in the binary program, we need to traverse all valid execution paths one by one. For this purpose, we used a symbolic execution platform called S2E. Among other symbolic execution platforms, we chose S2E since it runs on binary programs, supports Linux and Windows platforms, and provides detailed documentation. Though our analysis does not cover Linux binaries, we also aim to support Linux systems in the future.

Before we run the symbolic execution, we configure S2E's environment. First, we specify the starting point of the execution since we do not want S2E to analyze the binary program as it is. Instead, we make S2E to run our injector application, then, the application runs the binary program after it replaces API functions with the modeled ones. Next, we disable some of the default plugins brought by the S2E in order not to slow our analysis down [57].

After the configuration, we run S2E who runs the injector which executes the binary program that is linked to our function models, symbolically. During the symbolic execution, S2E traverses the program branches according to detail level, i.e. using concrete or symbolic return values.

If all function models return symbolic values, S2E traverses all the branches. Otherwise, it traverses only a subset of the total branches with respect to the function return value. When S2E visits a function model in a branch, the model creates an execution trace without calling any other Windows API functions. S2E stops after it visits all the possible branches or the user terminates the execution.

S2E helped our analysis by providing an execution environment where we can collect all function traces to extract function call sequences of the binary program. Since it runs the program symbolically, it traverses all possible branches and invokes the function models in all possible combinations. When a function model returns a symbolic value, S2E marks the memory area of the value as symbolic. Whenever this memory area is used in a statement, S2E creates, if not exists, a path condition by keeping the symbolic value as unknown. If the statement is a control statement, such as an if statement or a for loop, it forks a new execution branch and duplicates the path condition and updates with the new condition.

For example, in Figure 6 GetRandomInteger function returns a symbolic value and this value is used in conditional statements <A> and <B>. When S2E reaches the statement in <A> it forks the branch execution and duplicates the current path condition which is empty right now. Then, it appends the path condition of the left branch with the mathematical equation that satisfies the condition. If the condition is mathematically correct, it continues to execute the branch. Otherwise, it stops the execution. Similarly, S2E appends the right branch's path condition with the unsatisfying condition and

checks the satisfiability of the equation. When S2E reaches statement <B>, it forks the branch execution again, duplicates the current path condition, and appends them with corresponding conditions.

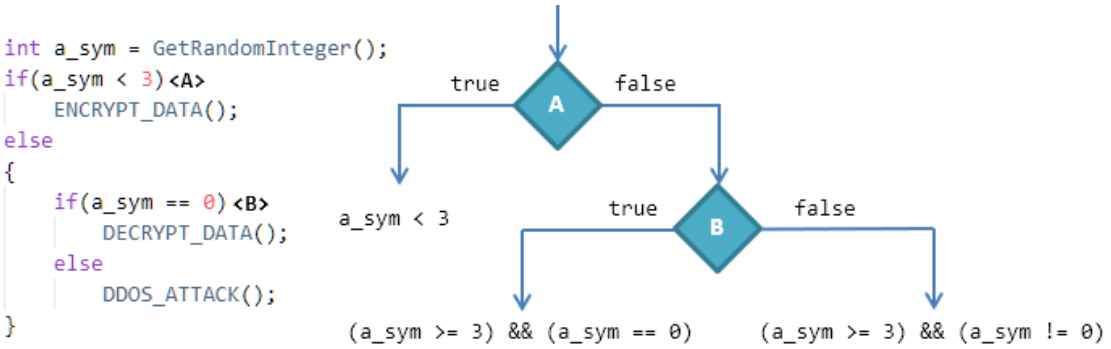

Figure 6 : Sample Path Condition Generation

As a result, whenever a function model returns a symbolic value, S2E runs symbolic execution to discover all possible branches. So, it makes our toolset capable of discovering even the hidden execution branches where it uses their function traces to extract hidden call sequences. In other words, if all function models return symbolic values, our methodology discovers all possible call sequences. However, to avoid the state space explosion problem, we do not recommend all functions to return symbolic values. Users may select certain functions to return desired concrete values using detail level input after seeing the exact values of symbolic variables. So, users may keep the balance between path discovery and symbolic execution performance.

After symbolic execution, our analysis parser module processes the execution log of S2E including the trace generated during the dynamic symbolic execution. This module extracts three kinds of information that we are interested in the execution output.

The first one is fork information. Whenever S2E reaches a control statement, it forks another execution path when the statement depends on a symbolic value and logs it to the output. Although we trigger forking via our function models to discover hidden program branches, it complicates the traceability of the execution. Therefore, it is essential to use fork information to trace S2E output in the correct order; so, our analysis parser extracts this information to make sense of other output information.

The second one is function traces. Every model emits its function trace whenever S2E invokes it. Our parser extract function traces to create function call sequences of the binary application. However, function traces are not meaningful by themselves since

S2E does not invoke functions in branch order. Therefore, our parser uses fork information to put function traces in order and creates the function call sequences.

The third one is the function evidences. Even though they are generated in a similar way with function traces, their form differs from them as we show in Figure 4. Therefore, we handled them separately.

Our analysis parser starts with discarding all the unnecessary data from the output of S2E. That is, it discards all the information other than fork, function traces and function evidences. Afterward, it uses fork information to put function traces in order and it achieves complete function call sequences for different program paths. Then, it also puts function evidences in order and finishes its job.

3.6 Analyzing the Call Sequence

In order to detect malicious sequences in the binary program, we create a module using Python called log analyzer. Log analyzer reads each call sequence of the binary program, extracted by analysis parser, and compares it with respect to the malicious function sequence, provided by the user. The log analyzer tries to match each malicious call sequence function in the correct order inside the program call sequence using a regular expression match. Meanwhile, in order to eliminate a sequence hiding attempt, our analyzer neglects irrelevant function calls inside the program call sequence while it is looking for the next malicious function.

```
OpenProcess, VirtualAllocEx, GetCommandLineA, GetFileSize, WriteProcessMemory, CreateRemoteThread
```

Figure 7 : Sample Program Call Sequence

For example, in order to satisfy the requirement of the DLLInjection sequence, shown in Figure 2, the program binary should have a call sequence containing all the functions of the malicious sequence in the given order. Recall that the sequence is specified as DLLInjection, OpenProcess, VirtualAllocEx, WriteProcessMemory, CreateRemoteThread where the first element is the name of the attack. However, a call sequence of the binary program may be in the form shown in Figure 7. In this case, our algorithm starts by searching for the OpenProcess function inside the call sequence of the binary program. If the algorithm finds a match, it continues to search for the next function, VirtualAllocEx. After matching the VirtualAllocEx function, the log analyzer tries to find the next one WriteProcessMemory. However, the call sequence of the binary program contains GetCommandLineA and GetFileSize functions before the WriteProcessMemory function. In such a case, our algorithm discards these unexpected functions as they may be put in order to hide the malicious sequence. Also, there would

be a repetition in the call sequence of the binary program. For example, instead of the GetCommandLineA function, there could be another VirtualAllocEx function. Then, our algorithm also ignores the repetition as it searches for the WriteProcessMemory function. Finally, our algorithm finds the match for the CreateRemoteThread function inside the call sequence. When our log analyzer matches all the functions inside the malicious sequence input, it warns the user and stops the analysis. In the result, it shows the name of the matching sequence, DLLInjection.

In some cases, the malicious call sequence input may contain pipe symbols '|' to indicate function variance. Whenever our log analyzer module encounters this symbol, it accepts the function either on the left or right side of the pipe symbol as a match. For example, if the malicious call sequence input is given as in Figure 8, our log analyzer module again warns the user if it matches with the function GetFileSize instead of the WriteProcessMemory function.

```
DLLInjection,OpenProcess,VirtualAllocEx,GetFileSize|WriteProcessMemory,CreateRemoteThread
```

Figure 8 : A Sample Malicious Call Sequence with a Pipe Symbol

3.7 Displaying Evidences

In order to support the results of the call sequence analysis, our log analyzer module also shows function evidences for the program paths containing malicious function sequences. So, even though our tool set produces a false-positive result, it supports the decision by displaying evidences. In this way, the user avoids making false decisions since evidences consolidate the analysis result by bringing the power of manual investigation. Furthermore, evidences accelerate early iterations of the analysis. That is, it guides users to decide whether they need to increase the detail level of the analysis or not.

Figure 9 shows a sample program for displaying evidences and a sample for generated function models are shown in Figure 10. The program in Figure 9 starts with a variable declaration of dayOfMonth in line 3. Then, the program calls Function_A with the parameter 60000 and uses dayOfMonth variable to call Function_B. Next, if Function_B sets the variable value to 15, the program calls Function_C with a string value "C:/" otherwise program ends with status value 0.

```c
int main()
{
    int dayOfMonth;
    Function_A(60000);
    Function_B(&dayOfMonth);
    if(dayOfMonth == 15)
        Function_C("C:/");
    return 0;
}
```

Figure 9 : Sample Program for Displaying Evidences

```c
static void Function_A_Model(_In_ int millis)
{
    Message("Function_A\n");
    Message("[Function_A|evidence] {[Wait for %d (ms)]}\n",millis);
    return;
}

static void Function_B_Model(_Out_ int* dayOfMonth)
{
    Message("Function_B\n");
    int retVal = 1;
    if(detail.compare("Function_B" == 0)){
        S2EMakeSymbolic(&dayOfMonth, sizeof(int), "Function_B");
    }else{
        *dayOfMonth = retVal;
    }
    return;
}

static int Function_C_Model(_In_ char* dir)
{
    Message("Function_C\n");
    Message("[Function_C|evidence] {[Deleting directory %s]}\n",dir);

    int retVal = 1;
    if(detail.compare("Function_C" == 0)){
        S2EMakeSymbolic(&dayOfMonth, sizeof(int), "Function_C");
    }else{
        /* concrete return */
    }
    return retVal;
}
```

Figure 10 : Sample Function Models for Displaying Evidences

During the analysis of the sample program shown in Figure 9, assume that only Function_B is configured to return symbolic value. The evidence output generated for this program is shown in Figure 11. In the output, we display 2 types of evidences: function evidences and execution evidences. The function evidences start after the EVIDENCES :: tag and their content consist of the function names encountered along the execution path and their input arguments. Our analyzer module presents the function evidences in human-readable form as they are emitted by the function models. Execution evidences are the rest of the data shown in Figure 11 which our log analyzer combines in the end of the analysis.

The first line of Figure 11 shows the analyzed call sequence name. Then, the log analyzer displays a warning message by giving the encountered malicious input sequence. Next, our log analyzer shows the complete malicious sequence. In the EVIDENCES:: section, the log analyzer displays the function evidences as they are created by the function models. Each line starts with the name of the function model given in the square brackets. Then, evidence information follows in curly brackets. For example, the first function evidence indicates that the function model of Function_A is called with a value that suspends the execution by 60000 milliseconds. Similarly, second function evidence indicates that the function model of Function_C deletes the directory "C:/".

After the TestCaseGenerator tag, our log analyzer displays symbolic return values of the model functions in little-endian byte order and ASCII formats. The meaning of the byte fields strongly depends on the function's return type. In our sample, Function_B returns an integer value symbolically with the length of 4 bytes. The value is set to 15 in decimal. Detailed usage of a symbolic return value is explained in section 4.2.2.

```
Analyzing the call sequence ending with [State X]

WARNING :: Seq_XYZ is encountered.

SEQUENCE :: Seq_XYZ, Function_B | Function_C

EVIDENCES ::
[Function_A|evidence] {[Wait for 60000 (ms)]}
[Function_C|evidence] {[Deleting directory C:/]}

TestCaseGenerator:
    Function_B = {0x0, 0x0, 0x0, 0xe}; (string) "...."
```

Figure 11 : Sample Evidence Output

3.8 Model Refining

After examining the evidences, users may want to perform future analysis on the binary program to achieve better results. In this case, our methodology supports users to re-analyze the program using different settings. For example, if the analysis stops after executing a certain function, users may modify the default return values shown in Table 2 or they may modify the detail level of the analysis to use symbolic return values rather than the concrete ones. So, our methodology allows users to analyze the binary program iteratively to achieve better results by providing a configurable toolset.

CHAPTER 4

EXPERIMENTS

In this chapter, we show our experimental work on our methodology. First, we present the experimental setup where we conduct our experiments. Next, we show the effectiveness of our methodology by analyzing a synthetic and an actual malware.

4.1 Experimental Setup

We evaluated our methodology on a virtual machine running on a desktop with a 3.30 GHz Intel(R) Core(TM) i5-6600 CPU and 32GB of RAM. The virtual machine had 16GB RAM and was running Ubuntu 20.04.1. Besides, the virtual machine performed symbolic execution on S2E's QEMU environment running Windows 10 Pro 1909 x86_64.

In order to evaluate our methodology, we implemented our modules in a combination of C++ and Python. We developed 6 modules namely: model generator, library finder, analysis parser, log analyzer, a DLL file having function models, and injector application. We wrote the DLL file and injector modules in C++ and they are around 2500 lines of code. On the other hand, we implement the rest of the modules in Python and they consist of about 600 lines of code.

4.2 Experiment 1: Synthetic Malware

4.2.1 Synthetic Malware

In order to show the capabilities of our methodology, wrote a synthetic malware and analyzed it. Our malware uses two techniques to hide its malicious activity: system time discovery [58] and sandbox evasion [59]. Then, it performs DLL injection using a DLL file we wrote.

Malware such as Friday 13th [60], Chernobyl [61], and FatDuke [62] use system time discovery techniques to prevent detection by delaying its malicious behavior until a specified time which is also called a time-bomb. In our malware, we also used this technique to show that our methodology can reveal hidden malware behavior, hidden program branches, and analyze all possible call sequences to detect a malicious sequence in program binary. Figure 12 shows the corresponding code segment. We programmed our malware so that it performs its malicious activity on 10th of November 2040. In Figure 12, detecting a time bomb appears to be a straightforward process since we can access the source code of the malware. However, it is a highly complex task to detect such a code segment in a binary program using obfuscation methods such as packaging and encryption.

We also add a system check technique to perform sandbox evasion [59]. It is a hiding technique that malware such as Astaroth, Evilnum, MegaCortex, and RogueRobin [59] uses to conceal its malicious behavior if the malware infers that it is under analysis. The malware checks system artifacts associated with the sandbox environment, such as device names, available memory, and CPU core, to evade it. So, we put a CPU core count control in our malware to show that our methodology can collect the call sequences hidden behind sandbox evasion. Figure 13 shows our malware's code snippet performing sandbox evasion. In the code, we allowed malware to activate if the target device has four or more CPU cores.

After using system time discovery and sandbox evasion techniques our synthetic malware performs its malicious behavior, process injection. Process injection is the technique that malware injects arbitrary code into a live separate process in order to make the live process perform the malicious activity. So, malware evades from defense mechanisms, such as anti-viruses, and access privileges of the live process. In order to show that our methodology is capable of detecting a malware technique that a signature-based system cannot discover, we used it in our experiments.

```
/*
* Setup a Time Bomb
*/
_SYSTEMTIME lTime;
GetLocalTime(&lTime);
if (lTime.wYear == 2040)
    if (lTime.wMonth == 11)
        if (lTime.wDay == 10)
            (void)0;// Bomb is activated
        else
            exit(-1); // day is wrong
    else
        exit(-1); // month is wrong
else
    exit(-1); // year is wrong
```

Figure 12 : Time Bomb Code Snippet

```
/*
* Perform sandbox evasion
*/
SYSTEM_INFO sysinfo;
GetSystemInfo(&sysinfo);
int numCPU = sysinfo.dwNumberOfProcessors;
if (numCPU >= 4)
    (void)0;// it is not a sandbox
else
    exit(-1); // sandbox or old system
```

Figure 13 : Sandbox Evasion Code Snippet

According to Mitre ATT&CK [63] process injection has 11 sub-techniques and we used the dynamic-link library (DLL) injection technique in our malware [64]. In this technique, the malware performs Windows API calls to inject a DLL file into a separate live process. First, the malware injects the path of the DLL in the address space of the process. Then, malware invokes a new thread to load the DLL and the new thread runs the code, the malicious activity, inside the DLL. As a result, the process performs the malicious activity with its privileges and malware stays hidden. As we already know the malicious call sequence of a DLL injection technique, shown in Figure 2, we decided to use it in our experiment and we used the data in Figure 2 as input to our experiment. The source code of our malware is given in APPENDIX A.

4.2.2 Experiment1 and Results

In the analysis, the decompiler found 15 dynamically linked functions. However, our toolset only modeled 9 of them as our methodology is only interested in the functions that belong to Windows API and these functions cover 100% of the Windows API functions used in the binary program. Table 3 shows dynamically linked functions extracted by the decompiler. Then, our model generator automatically creates models for the chosen functions. In total, around 150 lines of C++ code are created automatically. Even though our decompiler did not extract it, we modeled the ExitProcess function of the Windows API to increase the traceability of our call sequences.

To avoid state space explosion, we make all function models return concrete values at the beginning of the analysis. However, when we analyzed the function call sequences, we realized that the malware did not call any function after certain ones. For example GetLocalTime, GetSystemInfo and GetModuleHandleW functions were the last functions that we detected in call sequences. Therefore, we used expert opinion to change the default return values we show in Table 2. Then, we regenerate the models and run the analysis again to discover new paths. Furthermore, we modified the detail level of the analysis so that the functions returning simple C structs, such as GetLocalTime and GetSystemInfo, return symbolic values.

Table 3: Synthetic Malware Dynamically Linked Functions

Function Name	Model		Function Name	Model
_errno	No		GetSystemInfo	Yes
CloseHandle	Yes		Memcpy	No
CreateRemoteThread	Yes		memmove	No
exit	No		OpenProcess	Yes
free	No		VirtualAllocEx	Yes
GetLocalTime	Yes		wcstol	No
GetModuleHandleW	Yes		WriteProcessMemory	Yes
GetProcAddress	Yes			

Depending on the complexity of the binary program and the number of symbolic variables, the symbolic execution may take long hours. In order to avoid state space explosion and see the effects of modifications as soon as possible, such as detail level, we put a time limit for the symbolic execution. So, the symbolic execution engine run

until either execution finishes or the timer for execution expires. During the experiment, we saw that approximately 10 minutes of execution provide enough information to conclude the analysis.

Our log analyzer extracted the malicious call sequence given in Figure 14. In this figure, each line corresponds to a model invocation. The first element in a line shows the elapsed time, in seconds since the symbolic execution started. The second element shows the state number during the symbolic execution and the third one shows the invoked function model name.

```
162 [State 0] GetLocalTime_Model
182 [State 3] GetSystemInfo_Model
190 [State 4] OpenProcess_Model
190 [State 4] GetModuleHandleW_Model
190 [State 4] GetProcAddress_Model
190 [State 4] VirtualAllocEx_Model
190 [State 4] WriteProcessMemory_Model
190 [State 4] CreateRemoteThread_Model
190 [State 4] CloseHandle_Model
```

Figure 14 : Synthetic Malware Malicious Call Sequence

```
/*
* Setup a Time Bomb
*/
_SYSTEMTIME lTime;
GetLocalTime(&lTime);
if (lTime.wYear == 2040)      <A>
    if (lTime.wMonth == 11)   <B>
        if (lTime.wDay == 10) <C>
            (void)0;// Bomb is activated
        else
            exit(-1); // day is wrong
    else
        exit(-1); // month is wrong
else
    exit(-1); // year is wrong
/*
* Perform sandbox evasion
*/
SYSTEM_INFO sysinfo;
GetSystemInfo(&sysinfo);
int numCPU = sysinfo.dwNumberOfProcessors;
if (numCPU >= 4) <D>
    (void)0;// it is not a sandbox
else
    exit(-1); // sandbox or old system
```

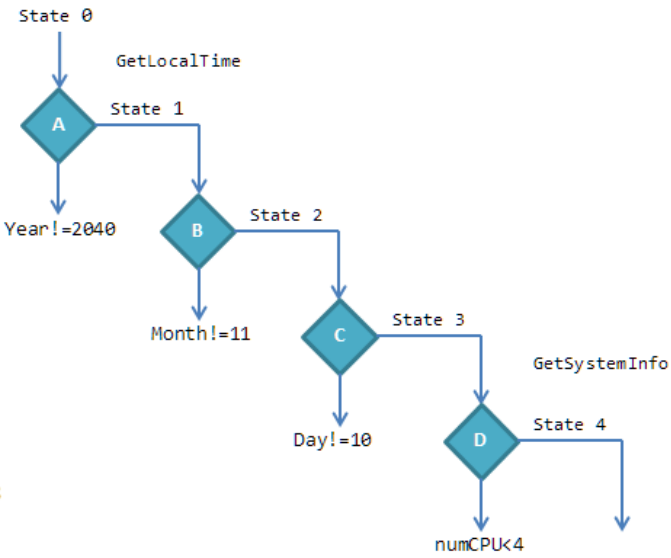

Figure 15 : Symbolic Execution States of Synthetic Malware

In Figure 15, we showed the symbolic state creation of our malware. We mark each decision point with < and > signs in the code and put a diamond shape for its program flow performed by the symbolic execution engine. Our log analyzer module creates the first two lines of the call sequence, in Figure 14, according to the flow we show in Figure 15. Then, it extracts the rest of the call sequence since the symbolic engine calls our model functions according to our malware's activity we show in APPENDIX A.

To detect the DLLInjection attack, we used a malicious call sequence input in the literature [16] where we show in Figure 2. Then, our log analyzer module detected the malicious sequence and generated the results in Figure 16. The first line in the result shows the analysis step. In this case, our log analyzer module analyzes the call sequence in Figure 14. Then, it prints a warning message that DLLInjection sequence is detected. Next, it prints the malicious sequence of DLLInjection and shows the evidences.

```
Analyzing the call sequence ending with [State 4]

WARNING :: DLLInjection is encountered.

SEQUENCE :: DLLInjection, OpenProcess, VirtualAllocEx,
            WriteProcessMemory, CreateRemoteThread

EVIDENCES ::
[OpenProcess|evidence] {[Open process with ID: 123]}
[GetModuleHandleW|evidence] {[Get the handle for the module: 'kernel32.dll']}
[GetProcAddress|evidence] {[Retrieve the function: 'LoadLibraryA' from a DLL file]}
[VirtualAllocEx|evidence] {[Allocation size: 13, allocation type: 0x3000, protections: 0x4]}
[WriteProcessMemory|evidence] {[ Data written to the process memory:
    0x6d 0x61 0x6c 0x69 0x63 0x69 0x6f 0x75 0x73 0x2e 0x64 0x6c 0x6c
    in ASCII format "malicious.dll"]}
TestCaseGenerator:
    v0_LocalTime_0 = {0xf8, 0x7, 0xb, 0x0, 0x0, 0x0, 0xa,
    0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}; (string) "................"
    v1_SystemInfo_1 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
    0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x80, 0x0, 0x0, 0x0, 0x0,
    0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
    (string) "................................."
```

Figure 16 : Synthetic Malware Analysis Result

Our log analyzer displays the evidences after EVIDENCES tag of Figure 16. Each line starting with square brackets shows evidence created by a function model and corresponding evidence information is presented between the curly brackets. For example, the GetModuleHandleW function created evidence for the requested access to kernel32.dll module and the GetProcAddress function indicated the access request for the LoadLibraryA function.

Moreover, our log analyzer displayed the symbolic return values of the function models satisfying the program path that generates the call sequence in Figure 14. The return types of the functions that return symbolic data are shown in Figure 17 and Figure 18. In Figure 16, v0_LocalTime_0 represents the symbolic value of the GetLocalTime function model and its C++ struct is shown in Figure 17. The figure suggests that the first 2 bytes of the symbolic value, 0xf8 and 0x7, represent the wYear element of the _SYSTEMTIME struct. These bytes denote the year 2040 in little-endian format. Similarly, next 2 bytes, 0xb and 0x0, represents the wMonth element and it is 11 in decimal. Finally, 4th byte pair in v0_LocalTime_0 represents the wDay field and it is 10 in decimal.

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME, *LPSYSTEMTIME;
```

Figure 17 : Return Struct of GetLocalTime Function [70]

```
typedef struct _SYSTEM_INFO {
  union {
    DWORD dwOemId;
    struct {
      WORD wProcessorArchitecture;
      WORD wReserved;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
  DWORD     dwPageSize;
  LPVOID    lpMinimumApplicationAddress;
  LPVOID    lpMaximumApplicationAddress;
  DWORD_PTR dwActiveProcessorMask;
  DWORD     dwNumberOfProcessors;
  DWORD     dwProcessorType;
  DWORD     dwAllocationGranularity;
  WORD      wProcessorLevel;
  WORD      wProcessorRevision;
} SYSTEM_INFO, *LPSYSTEM_INFO;
```

Figure 18 : Return Struct of GetSystemInfo Function [71]

v1_SystemInfo_1 represents the symbolic value of GetSystemInfo function model and we show its C++ struct in Figure 18. The bytes between 21st and 24th represent the dwNumberOfProcessors field of the struct. During the analysis, the symbolic engine resolved the field as 128 in decimal which satisfies line 22 in Figure 15.

In the end, our methodology captured the DLL injection attack of our malware whose cyclomatic complexity is 11. The symbolic execution of the malware took 190 seconds with 2 symbolic return values and 4 program branches. Although our methodology achieved 50% branch and 69% line coverages, the toolset executed 100% of the Windows API functions invocations of the malware It is also possible to achieve 100% branch coverage by using detail level input, but it is not a cost-effective solution as the toolset already detect the malicious sequence with less symbolic variables.

After we examined the evidences, we concluded that our malware performs a DLL injection attack on November 10, 2040 if it is not running on a virtual machine. During the DLL injection attack, first, it opens a process whose ID is 123. Then, it retrieves the kernel32 library and loads the LoadLibraryA function. Next, it allocates the process memory for 13 bytes and inserts ‘malicious.dll’ text in the memory. Finally, it creates a remote thread. This experiment shows the detection capability of our methodology with respect to the malware using evasion techniques.

4.3 Experiment2: WannaCry

4.3.1 WannaCry Malware

In order to show that our methodology is also applicable to analyzing real-world problems, we analyzed WannaCry ransomware. As of its first report in May 2017, WannaCry has spread to more than 150 countries. It uses a Windows vulnerability, MS17-010, to gain access to the systems and it encrypts user files. Then, it demands Bitcoin worth $300 or $600 to decrypt the data [65]. Malware analysts identify the malware as it is composed of two components namely, worm and encryption. The initial component behaves as a package containing the encryption component. As a sandbox avoidance mechanism, it tries to access a web page. If it connects to the page successfully, it stops its malicious behavior. Otherwise, it extracts the encryption component from its resource and executes it. After that, the encryption component changes the file attributes in its directory and starts encryption [3] [66]. In this experiment, we run our analysis on the encryption component.

In order to make sure that we analyze the correct malware component, we calculated sha256 and md5 hashes of the malware. Then, we verified the calculated values using a previous study [66] and a malware database [67]. The md5sum value is calculated as

84c82835a5d21bbcf75a61706d8ab549 and the sha256sum value is calculated as ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa.

4.3.2 Experiment2 and Results

In the analysis, the decompiler found more than 105 dynamically linked functions and we are interested in 64 of them. Table 4 shows the chosen functions. Then, our model generator automatically creates models for the chosen functions and generates around 1100 lines of C++ code automatically.

Next, to analyze behavior, we create a malicious call sequence, shown in Figure 19, by using our previous experience in synthetic malware, explained 4.2. In this way, we set our log analyzer module to create a warning if any one of the WriteProcessMemory, LoadLibraryA, or GetProcAdress functions are invoked.

```
WannaCryAnalysis, WriteProcessMemory|LoadLibraryA|GetProcAddress
```

Figure 19 : Malicious Call Sequence Input of WannaCry Analysis

```
183 [State 0] GetStartupInfoA_Model
183 [State 0] GetModuleHandleA_Model
183 [State 0] GetModuleFileNameA_Model
183 [State 0] GetComputerNameW_Model
183 [State 0] SetCurrentDirectoryA_Model
183 [State 0] RegCreateKeyW_Model
183 [State 0] RegCreateKeyW_Model
183 [State 0] FindResourceA_Model
183 [State 0] CreateProcessA_Model
183 [State 0] WaitForSingleObject_Model
183 [State 0] TerminateProcess_Model
183 [State 0] GetExitCodeProcess_Model
183 [State 0] CloseHandle_Model
183 [State 0] CloseHandle_Model
183 [State 0] CreateProcessA_Model
183 [State 0] WaitForSingleObject_Model
183 [State 0] TerminateProcess_Model
183 [State 0] GetExitCodeProcess_Model
183 [State 0] CloseHandle_Model
183 [State 0] CloseHandle_Model
183 [State 0] LoadLibraryA_Model
183 [State 0] GetProcAddress_Model
183 [State 0] GetProcAddress_Model
183 [State 0] GetProcAddress_Model
183 [State 0] GetProcAddress_Model
183 [State 0] GetProcAddress_Model
183 [State 0] GetProcAddress_Model
```

Figure 20 : WannaCry Malicious Call Sequence

Table 4: Generated Function Models for WannaCry

CloseHandle	GetModuleHandleA	RegCloseKey
CloseServiceHandle	GetProcAddress	RegCreateKeyW
CopyFileA	GetProcessHeap	RegQueryValueExA
CreateDirectoryA	GetStartupInfoA	RegSetValueExA
CreateDirectoryW	GetTempPathW	SetCurrentDirectoryA
CreateFileA	GetWindowsDirectoryW	SetCurrentDirectoryW
CreateProcessA	GlobalAlloc	SetFileAttributesW
CreateServiceA	GlobalFree	SetFilePointer
CryptReleaseContext	HeapAlloc	SetFileTime
DeleteCriticalSection	HeapFree	SetLastError
EnterCriticalSection	InitializeCriticalSection	SizeofResource
FindResourceA	IsBadReadPtr	Sleep
FreeLibrary	LeaveCriticalSection	StartServiceA
GetComputerNameW	LoadLibraryA	SystemTimeToFileTime
GetCurrentDirectoryA	LoadResource	TerminateProcess
GetExitCodeProcess	LocalFileTimeToFileTime	VirtualAlloc
GetFileAttributesA	LockResource	VirtualFree
GetFileAttributesW	MultiByteToWideChar	VirtualProtect
GetFileSize	OpenMutexA	WaitForSingleObject
GetFileSizeEx	OpenSCManagerA	WriteFile
GetFullPathNameA	OpenServiceA	
GetModuleFileNameA	ReadFile	

Our log analyzer module discovered the call sequence shown in Figure 20. This sequence has the malicious call sequence we specified in Figure 19 and our log analyzer displayed evidences shown in Figure 21. Evidences show that, first, WannaCry gets the handle for itself as it passes the NULL parameter to the GetModuleHandleA function [68]. Then, it sets its current directory to its current directory and it creates a registry key with the name WannaCrypt0r under Software tab. Next, it hides all the files in its current directory by using 'attrib +h' command [69] and waits approximately 30 minutes. After it terminates an operation with a failure status, it grants full access to all the files in its current directory and below using 'icacls . /grant Everyone:F /T /C /Q', in directory'. Then it waits again around 30 minutes and terminates the process with a fail status. Lastly, it loads a library called 'advapi32.dll' and loads 6 functions responsible for encryption.

```
Analyzing the call sequence ending with [State 0]

WARNING :: WannaCryAnalysis is encountered.

SEQUENCE :: WannaCryAnalysis, RegCreateKeyW|LoadLibraryA|GetProcAddress

EVIDENCES ::
[GetModuleHandleA|evidence] {[Get module handle for (null)]}
[SetCurrentDirectoryA|evidence] {[Set current directory to 'c:\s2e\malware.exe']}
[RegCreateKeyW|evidence] {[Create a registry key with name: 'Software\WanaCrypt0r']}
[CreateProcessA|evidence] {[Create Process '(null)', with command line:
        'attrib +h .', in directory '(null)']}
[WaitForSingleObject|evidence] {[Wait for 1703624 (ms)]}
[TerminateProcess|evidence] {[Terminate the process with the code: 4294967295]}
[CreateProcessA|evidence] {[Create Process '(null)', with command line:
        'icacls . /grant Everyone:F /T /C /Q', in directory '(null)']}
[WaitForSingleObject|evidence] {[Wait for 1703624 (ms)]}
[TerminateProcess|evidence] {[Terminate the process with the code: 4294967295]}
[LoadLibraryA|evidence] {[Load the Library: advapi32.dll]}
[GetProcAddress|evidence] {[Retrieve the function: 'CryptAcquireContextA' from a DLL file]}
[GetProcAddress|evidence] {[Retrieve the function: 'CryptImportKey' from a DLL file]}
[GetProcAddress|evidence] {[Retrieve the function: 'CryptDestroyKey' from a DLL file]}
[GetProcAddress|evidence] {[Retrieve the function: 'CryptEncrypt' from a DLL file]}
[GetProcAddress|evidence] {[Retrieve the function: 'CryptDecrypt' from a DLL file]}
[GetProcAddress|evidence] {[Retrieve the function: 'CryptGenKey' from a DLL file]}
```

Figure 21 : WannaCry Analysis Result

When we examined the evidences, we concluded that WannaCry performs suspicious operations such as creating a registry key with an unusual name, hiding files, granting open accessibility for everyone, and importing encryption functions. Even though all of our function models return concrete values, our toolset invoked 25% of the function models in a single symbolic execution run and generate meaningful evidences for the malicious behavior. This shows the applicability of our methodology to a real-world problem.

4.4 Discussion

In the synthetic malware experiment, our methodology detected DLLInjection attack even though the malware uses evasion techniques, such as time discovery and sandbox evasion, to hide its malicious activity. In this way, we showed the effectiveness of our methodology with respect to the traditional behavioral and API analysis techniques. Our toolset found 100% of the Windows API functions using the decompiler and modeled these API functions using around 16 lines of C++ code per function model. As we used our expert knowledge to choose two functions that return symbolic values, our methodology captured the malicious call sequence even though the symbolic execution achieved 50% branch coverage and 69% line coverage. Furthermore, the symbolic execution invoked 100% of the modeled functions at least once. Also, as we chose the rest of the functions to return concrete values, the symbolic execution step only took 190 seconds and did not suffer from the state-space explosion problem. In the end, our toolset successfully found out that the synthetic malware attempts a DLLInjection attack on November 10, 2040 to a process whose ID is 123 if the running device has 8 CPU cores.

In the WannaCry experiment, we have shown that our methodology applies to a real-world problem. Our toolset found 105 dynamically linked functions models. According to [66], this number represents 91% of the total function imports. When we include the encryption functions that our toolset extracted during the symbolic execution the percentage rises to 97% although we only use concrete values during the analysis. Our toolset modeled 56% of the imported functions as it only modeled the Windows API functions. Then, generated 17 lines of C++ code per function model. As we only use concrete values during the symbolic execution, our toolset did not experience any state-space explosion problems and the symbolic execution took less than a second. Furthermore, we achieved to invoke 25% of modeled functions only using concrete values in a single symbolic execution by configuring the detail level input. In the end, our toolset explicitly displayed the evidences that the usage of Windows commands, such as attrib and icalcs, and the Windows API functions. Thus, it showed the users whether the program they were analyzing was behaving in an unexpected way. Our evidences also showed that, like the synthetic malware, WannaCry also uses time functions to delay its execution. Even though it does not use a time bomb, it delays the execution by an hour.

There studies, such as [4] and [16], using control flow diagrams and DNA squences to show their performance for detecting API call sequences. However, this information is insufficient to guide the user to better decisions. To the best of our knowledge, there are no studies that show human-readable evidences for the user to improve their analysis results and validate the decision made.

In summary, our methodology can detect malicious behavior behind time bombs and sandbox evasion techniques by using a malicious call sequence and symbolic variables. It is also applicable to a real-world problem even though we do have a sequence that is given in the literature and using only concrete values.

Assumptions. In our methodology, we assume that:

- Users have at least one known malicious API call sequence in advance to analyze a binary program.
- The malicious API call sequence does not commonly exist in benign software so that our methodology does not produce false-positive results.
- The binary program contains at least as many API calls as a malicious API call sequence to perform a reasonable analysis.
- The symbolic execution engine either calls the function models in a single execution thread or provides state information so that function models' execution order can be extracted.
- Function models are not forced to make actual Windows API calls so that the symbolic execution platform does not dive into the depths of system calls which may hinder symbolic execution performance.
- Users utilize evidence information to improve function models so that the symbolic execution platform does not suffer from state-space explosion problems.

Constraints. Even though our methodology provides a general solution for malware analysis, we create our toolset to show the feasibility of our approach and the toolset has the following constraints:

- It only analyzes 32-bit binary programs.
- It only supports the analysis of binary programs for Windows.
- It detects the malicious behavior only if the binary program contains a known malicious API call sequence.

CHAPTER 5

CONCLUSION

In this thesis, we develop a methodology for detecting malicious behavior in a binary program with API call sequence analysis using dynamic symbolic execution for the Windows platform. Using our methodology, we implement an extensible toolset that supports users to utilize the latest developments in the API call sequence literature. Also, we present a configurable API function modeling approach to avoid the state-space explosion problem of symbolic execution by enabling users to decide return values of the function models to either concrete or symbolic using detail levels.

In order to show the effectiveness of our methodology, we analyzed a synthetic malware performing DLL injection attack and a real-world malware called WannaCry. Our toolset generated more than 1200 lines of C++ code and modeled 75 Windows API functions for the analysis of these malware. In our experiments, we showed that our approach of combining function models and dynamic symbolic execution is a feasible way of detecting API call sequences of a given binary program. Also, we demonstrated the capability of our toolset by detecting a DLL Injection attack even though it is hidden behind obfuscation techniques such as time discovery and sandbox evasion. During the experiments, we also showed that our function models provide observable evidences for generating API call sequences to analyze a real-world problem. Our toolset successfully discovered a call sequence of WannaCry ransomware and generate evidences for its activities such as importing encryption functions, hiding files, granting file accessibilities, and creating registry keys.

Limitations. So far, our toolset is only capable of analyzing 32 bit Windows binary programs. Also, it does not have the capability of analyzing statically linked functions. Therefore, its function models are only limited to dynamically linked functions. In order to avoid state space explosion, our models support concrete return values. However, using concrete values may hinder the capability of detecting hidden branches. Moreover, our system is not capable of creating malicious function call sequences by

itself, instead, the user provides the sequence as an input. Also, our system detects malicious behavior if the call sequence of the behavior is already known.

Future Work. We plan to model a complete set of Windows API functions. So that, we can reduce one step from our methodology, dynamically linked function extraction. Furthermore, we are also interested in supporting 64 bit Windows programs, Linux systems, and statically linked functions in the future.

REFERENCES

[1] P. Okane, S. Sezer, and K. Mclaughlin, “Obfuscation: The Hidden Malware,” *Security & Privacy, IEEE*, vol. 9, pp. 41–47, May 2011, doi: 10.1109/MSP.2011.98.

[2] Ö. Aslan and R. Samet, “A Comprehensive Review on Malware Detection Approaches,” *IEEE Access*, vol. 8, p. 1, May 2020, doi: 10.1109/ACCESS.2019.2963724.

[3] LogRhythm Labs, “A Technical Analysis of WannaCry Ransomware.” May 2020. [Online]. Available: https://logrhythm.com/blog/a-technical-analysis-of-wannacry-ransomware/

[4] C. and L. Z. and N. J. and S. D. and Y. H. Brumley David and Hartwig, “Automatically Identifying Trigger-based Behavior in Malware,” in *Botnet Detection: Countering the Largest Security Threat*, C. and D. D. Lee Wenke and Wang, Ed. Boston, MA: Springer US, 2008, pp. 65–88. doi: 10.1007/978-0-387-68768-1_4.

[5] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu, “A Review of Android Malware Detection Approaches Based on Machine Learning,” *IEEE Access*, vol. PP, p. 1, May 2020, doi: 10.1109/ACCESS.2020.3006143.

[6] P. Sreekumari, “Malware Detection Techniques Based on Deep Learning,” May 2020, pp. 65–70. doi: 10.1109/BigDataSecurity-HPSC-IDS49724.2020.00023.

[7] Y. Liu, C. Tantithamthavorn, L. Li, and Y. Liu, “Deep Learning for Android Malware Defenses: a Systematic Literature Review.” May 2021.

[8] G. Pék, B. Bencsáth, B. Hu, and L. Buttyan, “nEther: In-guest Detection of Out-of-the-guest Malware Analyzers,” May 2011, doi: 10.1145/1972551.1972554.

[9] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed automated random testing,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 2005, vol. 40, pp. 213–223. doi: 10.1145/1065010.1065036.

[10] K. Sen, D. Marinov, and G. Agha, “CUTE: A concolic unit testing engine for C,” in *SIGSOFT Software Engineering Notes*, May 2005, vol. 30, pp. 263–272. doi: 10.1145/1095430.1081750.

[11] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” May 2008, vol. 8, pp. 209–224.

[12] P. Godefroid, M. Levin, and D. Molnar, “SAGE: Whitebox Fuzzing for Security Testing,” *ACM Queue*, vol. 10, p. 20, May 2012, doi: 10.1145/2093548.2093564.

[13] T. Avgerinos, S. Cha, B. Hao, and D. Brumley, “AEG: Automatic Exploit Generation.,” in *Communications of the ACM*, May 2011, vol. 57. doi: 10.1145/2560217.2560219.

[14] V. Chipounov, V. Kuznetsov, and G. Candea, “S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems,” *Computer Architecture News*, vol. 39, May 2012, doi: 10.1145/1961295.1950396.

[15] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel and G. Vigna, “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis,” May 2016, pp. 138–157. doi: 10.1109/SP.2016.17.

[16] Y. Ki, E. Kim, and H. K. Kim, “A Novel Approach to Detect Malware Based on API Call Sequence Analysis,” *International Journal of Distributed Sensor Networks*, vol. 2015, pp. 1–9, May 2015, doi: 10.1155/2015/659101.

[17] D. Rabadi and S. Teo, “Advanced Windows Methods on Malware Detection and Classification,” May 2020, pp. 54–68. doi: 10.1145/3427228.3427242.

[18] Microsoft, “Create C/C++ DLLs in Visual Studio.” May 2020. [Online]. Available: https://docs.microsoft.com/en-us/cpp/build/dlls-in-visual-cpp?redirectedfrom=MSDN&view=msvc-160

[19] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, 1st ed. No Starch Press, 2012.

[20] K. Hahn and K. Hahn, “Robust Static Analysis of Portable Executable Malware,” 2014.

[21] P. Szor, “The Art of Computer Virus Research and Defense,” May 2005.

[22] A. Moser, C. Kruegel, and E. Kirda, “Limits of Static Analysis for Malware Detection,” in *Proceedings - Annual Computer Security Applications Conference, ACSAC*, May 2008, pp. 421–430. doi: 10.1109/ACSAC.2007.21.

[23] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith, “Malware normalization,” May 2021.

[24] H. Borojerdi and M. Abadi, “MalHunter: Automatic generation of multiple behavioral signatures for polymorphic malware detection,” in *Proceedings of the 3rd International Conference on Computer and Knowledge Engineering, ICCKE 2013*, May 2013, pp. 430–436. doi: 10.1109/ICCKE.2013.6682867.

[25] Y. Tang, B. Xiao, and X. Lu, “Using a bioinformatics approach to generate accurate exploit-based signatures for polymorphic worms,” *Computers & Security*, vol. 28, pp. 827–842, May 2009, doi: 10.1016/j.cose.2009.06.003.

[26] M. Alzaylaee, S. Yerima, and S. Sezer, “DL-Droid: Deep learning based android malware detection using real devices.” May 2019.

[27] J. Booz, J. McGiff, W. Hatcher, W. Yu, J. Nguyen, and C. Lu, “Tuning Deep Learning Performance for Android Malware Detection,” May 2018, pp. 140–145. doi: 10.1109/SNPD.2018.8441128.

[28] A. Martín García, F. Fuentes, V. Naranjo, and D. Camacho, “Evolving Deep Neural Networks architectures for Android malware classification,” May 2017, pp. 1659–1666. doi: 10.1109/CEC.2017.7969501.

[29] N. McLaughlin, J. Martinez-Del-Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupe, and G. Ahn, “Deep Android Malware Detection,” May 2017, pp. 301–308. doi: 10.1145/3029806.3029823.

[30] E. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, “MalDozer: Automatic framework for android malware detection using deep learning,” *Digital Investigation*, vol. 24, pp. S48–S59, May 2018, doi: 10.1016/j.diin.2018.01.007.

[31] A. Pektaş and T. Acarman, “Deep learning for effective Android malware detection using API call graph embeddings,” *Soft Computing*, vol. 24, pp. 1–17, May 2020, doi: 10.1007/s00500-019-03940-5.

[32] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. Kumar, “Android malware detection based on system call sequences and LSTM,” *Multimedia Tools and Applications*, vol. 78, pp. 1–21, May 2019, doi: 10.1007/s11042-017-5104-0.

[33] A. Pektaş and T. Acarman, “Deep Learning To Detect Android Malware via Opcode Sequences,” *Neurocomputing*, vol. 396, May 2019, doi: 10.1016/j.neucom.2018.09.102.

[34] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, “Adversarial Examples for Malware Detection,” May 2017, pp. 62–79. doi: 10.1007/978-3-319-66399-9_4.

[35] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren, “Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection,” *IEEE Transactions on Information Forensics and Security*, vol. PP, p. 1, May 2019, doi: 10.1109/TIFS.2019.2932228.

[36] B. Kolosnjaji, A. Demontis, B. Biggio, and D. Maiorca, “Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables,” May 2018, pp. 533–537. doi: 10.23919/EUSIPCO.2018.8553214.

[37] S. Cesare and Y. Xiang, *Software similarity and classification*. 2012. doi: 10.1007/978-1-4471-2909-7.

[38] Norman Solutions, “Norman SandBox.” [Online]. Available: http://download01.norman.no/product_sheets/eng/SandBox_analyzer.pdf

[39] R. Paleari, L. Giampaolo, F. Roglia, and D. Bruschi, “A fistful of red-pills: How to automatically generate procedures to detect CPU emulators,” May 2009.

[40] V. Sathyanarayan, P. Kohli, and B. Bezawada, “Signature Generation and Detection of Malware Families,” May 2008, vol. 5107, pp. 336–349. doi: 10.1007/978-3-540-70500-0_25.

[41] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamzeh, “Malware detection based on mining API calls,” May 2010, pp. 1020–1025. doi: 10.1145/1774088.1774303.

[42] Y. Ye, D. Wang, T. Li, and D. Ye, “IMDS: Intelligent malware detection system,” May 2007, pp. 1043–1047. doi: 10.1145/1281192.1281308.

[43] R. Tian, M. R. Islam, L. Batten, and S. Versteeg, “Differentiating malware from cleanware using behavioural analysis,” in *Proceedings of the 5th IEEE*

International Conference on Malicious and Unwanted Software, Malware 2010, May 2010, pp. 23–30. doi: 10.1109/MALWARE.2010.5665796.

[44] M. Shankarapani, K. Kancherla, S. Ramammoorthy, R. Movva, and S. Mukkamala, "Kernel machines for malware classification and similarity analysis," May 2010, pp. 1–6. doi: 10.1109/IJCNN.2010.5596339.

[45] M. Shankarapani, S. Ramamoorthy, R. Movva, and S. Mukkamala, "Malware detection using assembly and API call sequences," *Journal in Computer Virology*, vol. 7, pp. 107–119, May 2011, doi: 10.1007/s11416-010-0141-5.

[46] H. Kim, M. Khoo, and Pietrolì, "Polymorphic Attacks against Sequence-based Software Birthmarks," May 2021.

[47] J. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, pp. 385–394, May 1976, doi: 10.1145/360248.360252.

[48] B. Chen, C. Havlicek, Z. Yang, K. Cong, R. Kannavara, and F. Xie, "CRETE: A Versatile Binary-Level Concolic Testing Framework," 2018, pp. 281–298. doi: 10.1007/978-3-319-89363-1_16.

[49] N. Stephens, J. Grosen, C. Salls, and A. Dutcher, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," May 2016. doi: 10.14722/ndss.2016.23368.

[50] S. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on Binary Code," pp. 380–394, May 2012, doi: 10.1109/SP.2012.31.

[51] R. Baldoni, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Assisting Malware Analysis with Symbolic Execution: A Case Study," May 2017, pp. 171–188. doi: 10.1007/978-3-319-60080-2_12.

[52] M. Alsaleh, J. Wei, E. Al-Shaer, and M. Ahmed, "gExtractor: Towards Automated Extraction of Malware Deception Parameters," May 2018, pp. 1–12. doi: 10.1145/3289239.3289244.

[53] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," in *Proceedings - IEEE Symposium on Security and Privacy*, May 2007, pp. 231–245. doi: 10.1109/SP.2007.17.

[54] L. Ďurfina, J. Křoustek, P. Matula, and P. Zemek, "A Novel Approach to Online Retargetable Machine-Code Decompilation," *Journal of Network and Innovative Computing*, vol. 2, pp. 224–232, May 2014.

[55] EasyHook, “Installing a remote hook using EasyHook with C++.” [Online]. Available: http://easyhook.github.io/tutorials/nativeremotehook.html

[56] Selenium, “SeleniumHQ Browser Automation.” [Online]. Available: https://www.selenium.dev/

[57] A. Herrera, “Analysing ‘Trigger-based’ Malware with S2E.” May 2018. [Online]. Available: https://adrianherrera.github.io/post/malware-s2e/

[58] MITRE ATT&CK®, “System Time Discovery, Technique T1124.” May 2021. [Online]. Available: https://attack.mitre.org/techniques/T1124/

[59] MITRE ATT&CK®, “Virtualization/Sandbox Evasion: System Checks, Sub-technique T1497.001.” May 2021. [Online]. Available: https://attack.mitre.org/techniques/T1497/001/

[60] Panda Security, “Friday 13th: Remembering one of the most infamous virus in history.” May 2018. [Online]. Available: https://www.pandasecurity.com/en/mediacenter/malware/famous-virus-history-friday-13th/

[61] Panda Security, “Chernobyl - Virus Information.” [Online]. Available: https://www.pandasecurity.com/en/security-info/2860/information/Chernobyl

[62] MITRE ATT&CK®, “FatDuke, Software S0512.” May 2020. [Online]. Available: https://attack.mitre.org/software/S0512/

[63] MITRE ATT&CK®, “MITRE ATT&CK®.” [Online]. Available: https://attack.mitre.org/

[64] MITRE ATT&CK®, “Process Injection: Dynamic-link Library Injection, Sub-technique T1055.001 .” May 2020. [Online]. Available: https://attack.mitre.org/techniques/T1055/001/

[65] Cybersecurity Infrastructure Security Agency (CISA)", “Indicators Associated With WannaCry Ransomware.” May 2018. [Online]. Available: https://us-cert.cisa.gov/ncas/alerts/TA17-132A/

[66] M. Akbanov and V. Vassilakis, “WannaCry Ransomware: Analysis of Infection, Persistence, Recovery Prevention and Propagation Mechanisms,” *Journal of Telecommunications and Information Technology*, vol. 1, pp. 113–124, May 2019, doi: 10.26636/jtit.2019.130218.

[67] Virus Total, “Virus Total Search.” [Online]. Available: https://www.virustotal.com/gui/home/search

[68] Microsoft, “GetModuleHandleA function (libloaderapi.h) - Win32 apps.” May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandlea

[69] Microsoft, “CreateProcessA function (processthreadsapi.h) - Win32 apps.” May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa

[70] Microsoft, “SYSTEMTIME (minwinbase.h) - Win32 apps.” May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-systemtime

[71] Microsoft, “SYSTEM_INFO (sysinfoapi.h) - Win32 apps.” May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/ns-sysinfoapi-system_info

APPENDICES

APPENDIX A

Synthetic Malware Source Code

```
#include "stdafx.h"
#include "Windows.h"
#include "stdio.h"
#include <string>

int _tmain(int argc, _TCHAR* argv[])
{
    char* dllPath = "malicious.dll";
    /*
    * Setup a Time Bomb
    */
    _SYSTEMTIME lTime;
    GetLocalTime(&lTime);
    if (lTime.wYear == 2040)
        if (lTime.wMonth == 11)
            if (lTime.wDay == 10)
                (void)0;// Bomb is activated
            else
                exit(-1); // day is wrong
        else
            exit(-1); // month is wrong
    else
        exit(-1); // year is wrong

```

```
/*
* Perform sandbox evasion
*/
SYSTEM_INFO sysinfo;
GetSystemInfo(&sysinfo);
int numCPU = sysinfo.dwNumberOfProcessors;
if (numCPU >= 4)
    (void)0;// it is not a sandbox
else
    exit(-1); // sandbox or old system

/*
* Access the target process whose PID is 123.
*/
HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, FALSE, 123);
if (process == NULL)
    exit(-1); // Could not open the process with given PID.
else
{
    /*
    * Access the LoadLibrary function.
    */
    LPVOID addr = (LPVOID)GetProcAddress(GetModuleHandle(L"kernel32.dll"),
                                         "LoadLibraryA");
    if (addr == NULL)
        exit(-1); // Could not load the LoadLibraryA function.
    else
    {
        /*
        * Allocate memory in the target process.
        */
        LPVOID arg = (LPVOID)VirtualAllocEx(process, NULL, strlen(dllPath),
                                            MEM_RESERVE | MEM_COMMIT,
                                            PAGE_READWRITE);
        if (arg == NULL)
            exit(-1); // Allocation failed.
        else
        {
            /*
            * Write the dll adress into the target process memory region.
            */
            int n = WriteProcessMemory(process, arg, dllPath,
                                       strlen(dllPath), NULL);
            if (n == FALSE)
                exit(-1); // Write failed.
```

```
70                  else
71                  {
72                      /*
73                      * Inject the DLL into the target process.
74                      */
75                      HANDLE threadID = CreateRemoteThread(process, NULL, 0, addr,
76                                                           arg, NULL, NULL);
77                      if (threadID == NULL)
78                          exit(-1); // Mission failed.
79                      else
80                          (void)0;  // Mission successful.
81
82                      /*
83                      * Close the handle.
84                      */
85                      CloseHandle(process);
86                  }
87              }
88          }
89      }
90      return 0;
91  }
```

TEZ İZİN FORMU / THESIS PERMISSION FORM

ENSTİTÜ / INSTITUTE

Fen Bilimleri Enstitüsü / Graduate School of Natural and Applied Sciences ☐

Sosyal Bilimler Enstitüsü / Graduate School of Social Sciences ☐

Uygulamalı Matematik Enstitüsü / Graduate School of Applied Mathematics ☐

Enformatik Enstitüsü / Graduate School of Informatics ☐

Deniz Bilimleri Enstitüsü / Graduate School of Marine Sciences ☐

YAZARIN / AUTHOR

Soyadı / Surname : .....
Adı / Name : .....
Bölümü / Department : .....

TEZİN ADI / TITLE OF THE THESIS (**İngilizce** / English) : .....
.....
.....
.....
.....

TEZİN TÜRÜ / DEGREE: **Yüksek Lisans** / Master ☐ **Doktora** / PhD ☐

1. **Tezin tamamı dünya çapında erişime açılacaktır. /** Release the entire work immediately for access worldwide. ☐
2. **Tez iki yıl süreyle erişime kapalı olacaktır.** / Secure the entire work for patent and/or proprietary purposes for a period of **two year. *** ☐
3. **Tez altı ay süreyle erişime kapalı olacaktır.** / Secure the entire work for period of **six months**. ***** ☐

***** *Enstitü Yönetim Kurulu Kararının basılı kopyası tezle birlikte kütüphaneye teslim edilecektir.*
A copy of the Decision of the Institute Administrative Committee will be delivered to the library together with the printed thesis.

Yazarın imzası / Signature ..... **Tarih** / Date .....