

國立臺灣大學管理學院資訊管理學研究所

碩士論文

Department of Information Management

College of Management

National Taiwan University

Master Thesis

運用生成式人工智慧協撰惡意程式行為分析敘事文本

Unleashing Malware Analysis and Understanding with
Generative AI

陳志剛

ZHI-KANG Chen

指導教授：孫雅麗 博士

Advisor: Yea-Li Sun, Ph.D.

中華民國 112 年 8 月

August, 2023





Acknowledgements

從中央大學資管系畢業之後，很幸運的來到了台大資管研究所，研究所生活的兩年轉瞬即逝，可能是內容十分充實吧？兩年就這樣悄悄地過去了。

在撰寫論文的過程中，我慢慢了解自己在「做學問」上的不足之處，以往我總是認為「實務」以及「實作」的能力是做重要的，畢竟這是最顯而易見得成果，也是職涯上最常見的評分標準；但是經過這兩年的研析，我慢慢體會到只有「動手做」的能力是不足的。

在此十分感謝我的論文指導教授：孫雅麗老師。老師總是能以較為宏觀的視野教導我，帶領我從另一個角度看待事情，並非我原本從「細節」上下手，直接去把程式碼寫出來，而是要從整體的「框架」去開始進行建構，過程中雖然我學習緩慢，但老師仍願意慢慢地引導我，多次將我從「細節的泥沼中」拉回來，重新審視我們真正該注重的事情。

此外也特別感謝中研院的陳孟彰老師、黃意婷老師，在多次共同會議中給予不同的意見、想法，讓我的論文能夠有更多發展的可能性。接著要感謝我們實驗室的各位同學，常常在沉悶氣氛中帶來歡樂的鄭亦辰、陪我研究學長系統的楊柏睿、以及常常跟我討論課堂上各種大小事的呂晟維同學，讓我的研究所生活十分豐富。這裡也要特別感謝所有從國小到研究所都沒有斷聯繫的朋友們，晚上大家在網路上互相抱怨、聊天，真的對我十分有幫助，是我在忙碌的研究所生活中一

個心靈的休憩點。

最後我要感謝我的家人們，家人們從未限制我想走的方向，讓我自由地進行探索，體諒我因課業的繁忙而常常無法參與家庭活動；感謝你們讓我知道我永遠有一個地方可以回去，讓我能無後顧之憂地往更高的境界前行。

準備脫離學生階段步入社會的我仍有許多不足之處，但我希望能在追求自己的理想之餘，慢慢回饋所有曾經幫助過我的人們，真的非常感謝在不同階段給予我支持的人們。





摘要

目前世界正處於一資訊爆炸的時代，資訊科技已經成為人類生活不可或缺的一部份，隨之而來的就是各種資安議題，其中惡意程式的分析更是一大項目，各種資安攻擊手法都離不開惡意程式的協助。現行在惡意程式行為的分析上常常使用所謂的動態分析，也就是將惡意程式至於一沙箱之中，透過側錄的方式取得其活動證據，這些活動證據常常由低階語言所組成。

許多資安公司對這些沙箱側錄結果都會產出技術報告，但是報告內容大都不容易消化，對於資訊安全管理者的威脅情報掌握與傳遞，或是一般人在資安領域的專業知識學習上，需要將這些報告轉成更容易理解、具有語意(以自然語言撰寫)的惡意程式技術摘要報告，較符合實際需求，但目前大多仰賴資安專家手動製作，缺乏有效率地產出方法。本論文的研究主要希望解決這種低階的結構語意難以理解的問題，預計會將 System Call Trace 的每一個詳細步驟進行語意上的轉換，將其從低階語言轉化為易於人類閱讀的高階語法，並且透過 ChatGPT 的輔助將其綜整成一篇文章，使研究人員能以最自然的方式快速地理解惡意程式的活動內容。

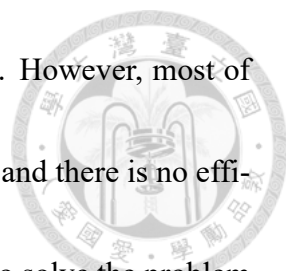
關鍵字：動態分析、攻擊情境圖、自然語言生成



Abstract

At present, the world is in an era of information explosion. Information technology has become an indispensable part of human life, and various information security issues follow. Among them, the analysis of malicious programs is a major project. Various information security attacks The methods are inseparable from the assistance of malicious programs. At present, the so-called dynamic analysis is often used in the analysis of malicious program behavior, that is, the malicious program is placed in a sandbox, and its activity evidence is obtained through the method of skimming, and the activity evidence is often composed of low-level language.

Many information security companies will produce technical reports on these sandbox skimming results, but most of the report content is not easy to digest. For information security managers to grasp and transmit threat intelligence, or for ordinary people to learn professional knowledge in the field of information security, These reports need to be converted into easier-to-understand, semantic (written in natural language) malicious program



technical summary reports, which are more in line with actual needs. However, most of them rely on information security experts to manually produce them, and there is no efficient way to produce them. The research of this paper mainly hopes to solve the problem that the low-level structural semantics are difficult to understand. It is expected that each detailed step of System Call Trace will be semantically converted from low-level language to high-level grammar that is easy for humans to read. , and integrated into an article with the assistance of ChatGPT, so that researchers can quickly understand the activities of malicious programs in the most natural way.

Keywords: Dynamically Analysis, Attack Scenario Graph, Natural Language Generation



Contents

	Page
Acknowledgements	i
摘要	iii
Abstract	iv
Contents	vi
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Objective	2
Chapter 2 Background & Related Work	3
2.1 Background	3
2.1.1 Execution Trace (System Call Trace)	3
2.1.2 Attack Scenario Graph	4
2.1.2.1 Source Node	4
2.1.2.2 Destination Node	4
2.1.2.3 Edge	6
2.1.2.4 Example	6
2.1.3 Algorithm	7
2.2 Related Work	9

2.2.1	Dynamic Malware Analysis in the Modern Era—A State of the Art Survey	9
2.2.2	HOLMES: Provenance Graph 與 Audit logs	9
2.2.3	Tactical Provenance Analysis for Endpoint Detection and Response Systems : Provenance Graph 與 API call logs	10
2.2.4	AttacKG: Constructing Technique Knowledge Graph from Cyber Threat Intelligence Reports	11
Chapter 3	Automated Narrative Essay Generation	13
3.1	System Architect	13
3.2	Issue: ASG Large Scale	14
3.2.1	For Node	14
3.2.2	For Edge	15
3.2.3	Reduction Result	16
3.3	Issue: System call 的低階結構、語義	18
3.4	Solution	18
3.4.1	Malware List of Syscall Steps	19
3.4.2	NL_Synonym Base	20
3.4.3	schema	20
3.4.4	Natural Language (NL) Transformer	21
3.4.4.1	Algorithm	22
3.5	案例分析: Dofloo	24
3.6	Issue: 自動化標註	25
3.7	Solution: ChatGPT Prompt	26
3.7.1	Discussion	28

3.7.1.1	Version	28
3.7.1.2	Limitation	28
3.7.2	Evaluation Output	28
3.8	案例分析: Gafgyt	29
3.9	案例分析: Darlloz	31
3.10	案例分析: Luabot	32
3.11	案例分析: Kaiten	34
Chapter 4	Conclusion	37
	List of Figures	38
	List of Tables	40
	References	41



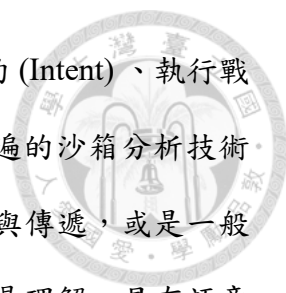


Chapter 1 Introduction

1.1 Motivation

想要深度瞭解一惡意程式的執行行為或活動、在語意研究上現行做法大多是利用動態分析 (dynamic analysis)，將程式放置在例如一沙箱 (sandbox) 裡執行，利用系統工具像是 strace 將其呼叫 system calls 錄製下來，產出 a system call trace，其詳細記錄惡意程式的實際執行過程，提供進一步的行為分析與瞭解。然而沙箱側錄之內容大多由作業系統層級或函式庫的函式呼叫 (system call, library API call) 組成，儘管這些都可能是惡意活動的具體操作證據，但皆為低階 (low-level) 的資料細節，要快速掌握並瞭解該惡意程式動機，以及其如何一步步地對目標系統進行駭侵、造成影響，須倚靠資安專業人士花費長時間作業，進行分析與人工判讀。此外，system call trace 資料量可能是龐大的 (數千個 system call invocations)，這對在數位鑑識的專家或是學生想要深入瞭解該惡意程式的執行活動、辨識其攻擊行為與惡意企圖，就會因為 system call 的低階程式結構與語義，難以看懂與掌握重要行為與執行企圖 (intent)。

目前網路上能找到知名資安公司 (Microsoft, Trend-Micro, Kaspersky, etc.) 會對某些惡意程式沙箱側錄結果撰寫技術報告 (technical report)，雖然具備參考價值，然而這些報告格式不一、內容紛雜，且大多僅針對惡意程式表現出的單獨活動提出警告，例如指出程式產生哪些可疑檔案、執行哪些高風險操作等…，缺乏從宏



觀角度探討惡意程式的整體活動，尤其忽略了該惡意程式之目的 (Intent)、執行戰略 (Tactic)、手法 (Technique) 等高階 (high-level) 描述資訊。普遍的沙箱分析技術報告內容都不容易消化，對於資訊安全管理者的威脅情報掌握與傳遞，或是一般人在資安領域的專業知識學習上，會需要將這些報告轉成更容易理解、具有語意 (以自然語言撰寫) 的惡意程式技術摘要報告 (technical summary report)，較符合實際需求，以協助對資安事件的判讀，目前大多仰賴資安專家手動製作，缺乏有效率地產出方法。

面對當前數位時代大量且快速變化的威脅，全世界資訊安全業界及學術研究正致力於找到新的可疑活動偵查與預警方法，此方法須能整合低階執行序具體威脅資訊，進一步產出高階語意技術分析摘要，同時具備易於理解、快速反應之特性。因此，如果可以將一惡意程式的重要惡意行為轉換成高階的自然語言，並且能夠將執行步驟做綜整並重點提示，將會對要瞭解惡意程式的人士，提供一個非常好的工具。本論文就是針對這些議題提出依據惡意程式執行行為圖，並自動化產生高階自然語言惡意程式行為敘事文本的研究成果。

1.2 Objective

利用所設計的 National Language Transformer 將 system call trace 中的每一個 system call invocation 針對其語義轉換成對應的自然語言動詞，以及將其作用的物件例如某一檔案，轉換成自然語言的受詞，同時並保留原始檔案名稱或路徑。再利用 ChatGTP 工具，成功地將轉譯後的簡易自然語言表示的執行步驟，以敘事 (narrative) 的方式呈現，綜整出容易閱讀的文章。



Chapter 2 Background & Related Work

2.1 Background

2.1.1 Execution Trace (System Call Trace)

Execution Trace 是用來表達 program 在系統底層運作的真實狀態，在 Windows 系統中可能以 API Call Trace 來表示，而在 Linux 會以 System Call Trace 來表示，包含三大要素：「1. system call、2. parameter、3. return value」。

例如：想要從讀取一個檔案的內容，System Call Trace 可能會長這樣：「read(3, "Hello World", 1024) = 0」，其中"read()"代表 system call，也就是其執行的動作 (action)；(3, "Hello World", 1024) 代表 parameter，此組參數在這個案例中表示讀取 File Descriptor = 3 的檔案，讀取到的內容為"Hello World"，Buffer 最多可以讀取 1024 Bytes；= 0 代表 return value，通常 0 代表正常運行、其他數值依照 system call 的設計可能有不同含意。

由上述的描述可知，System Call Trace 可以用來描述一個 program 在系統底層詳細的運作流程，我們可以透過這種方式來剖析惡意程式，進而客觀的理解其惡意行為。



2.1.2 Attack Scenario Graph

Attack Scenario Graph (ASG) 參考論文 [1, 4, 5] Graph 的設計進行建構，Graph Node 的部分主要參考了 1. File、2.Process、3.Network、4.Memory 類別，其餘未納入的類別原因會於章節”Related Work”逐一詳述。我們經過實際分析側錄結果後發現這 4 個類別仍有不足之處，因此擴增了 5 個類別，分別是 5.ID、6.Permission、7.Exit status、8.Timestamp、9.Resource Bytes，會於接下來介紹各類型的 Node 時詳細解釋原因。Graph 的組成可分為三個部分：

2.1.2.1 Source Node

Source Node 代表執行 System call 的主體，依據我們實際側錄的結果，總共有 2 種類別：

- File : System call 主體為 file descriptor、path 的類別，例如”/etc/rc.local/、/usr/local/bin/sed/” 這些檔案形式的物件。
- Process : System call 主體 process、command 的類別，例如執行 sed 這個指令後產生的 process，node 上就會以 sed 表示。

2.1.2.2 Destination Node

Destination Node 代表 System call 作用的受體，依據我們實際側錄的結果共有 9 種類別：

- File : System call 受體為 file descriptor、path 的類別，例如”/etc/rc.local/、/usr/local/bin/sed/” 這些檔案形式的物件。



- Process : System call 受體為 process、command 的類別，例如執行 sed 這個指令後產生的 process，node 上就會以 sed 表示。
- Network : System call 受體為 IP、NIC 的類別，例如”23.236.66.13:50050”這種 socket IP 位址、或是 eth0 這種網路介面卡的物件。
- Memory : System call 受體為 Memory Address 的類別，例如”0x8fe7000”這種記憶體位址。
- ID : System call 受體為 UID、GID、PID 的類別，例如”uid = 0、gid = 101、pid = 1520”，用來表示 user、group、process 的一個辨識符號。此為其他論文中未提及的一個類別，比較相似的類別只有論文 [5]HOLMES 中提及的 User 類別，我們認為以 ID 類別同時包含 user、group、process 辨識符的訊息，能提供更豐富的資訊，因此新增此一類別。
- Permission : System call 受體為 permission 的類別，此為其他論文中未提及的一個類別，我們在實際側錄惡意程式時發現某些 system call 會用來取得存取權限，例如 umask() 這一 system call 就會以 3 個數字來代表權限，以”000”為例就代表權限為可讀可寫可執行，因此新增此一類別。
- Timestamp : System call 受體為時間，此為其他論文中未提及的一個類別，我們在實際側錄惡意程式時發現惡意程式會不定期的使用某些 system call 來取得時間戳記，例如: 使用 time() 這一 system call 來取得當前時間戳記，因此我們新增一個類別來存儲這一資訊。
- Exit status : System call 受體為 process 結束的狀態，此為其他論文中未提及的一個類別，我們在實際側錄惡意程式時發現惡意程式欲結束時都會使用 exit() 類型的 system call 來進行離開的動作，其中會以一個數字來代表它的

離開狀態，例如”0”代表正常離開、”1”代表有異常狀況導致程序中斷，因此我們新增一個類別來存儲這一資訊。

- Resource Bytes：System call 受體為可用資源大小 (以 bytes 計算)，此為其他論文中未提及的一個類別，我們在實際側錄惡意程式時發現惡意程式會使用某些 system call 來取得自身可用資源的資訊，例如使用 prlimit() 這一 system call 可以得知當前的 process 會有多少資源可以使用，會得到如 8192*1024 bytes 這種資訊，因此我們新增一個類別來存儲這一資訊。

2.1.2.3 Edge

代表 System call 的 function name，例如: clone()、open()、read()、write()，用來表示惡意程式的執行動作，這一處理方式與論文 [1, 5] 相同。

2.1.2.4 Example

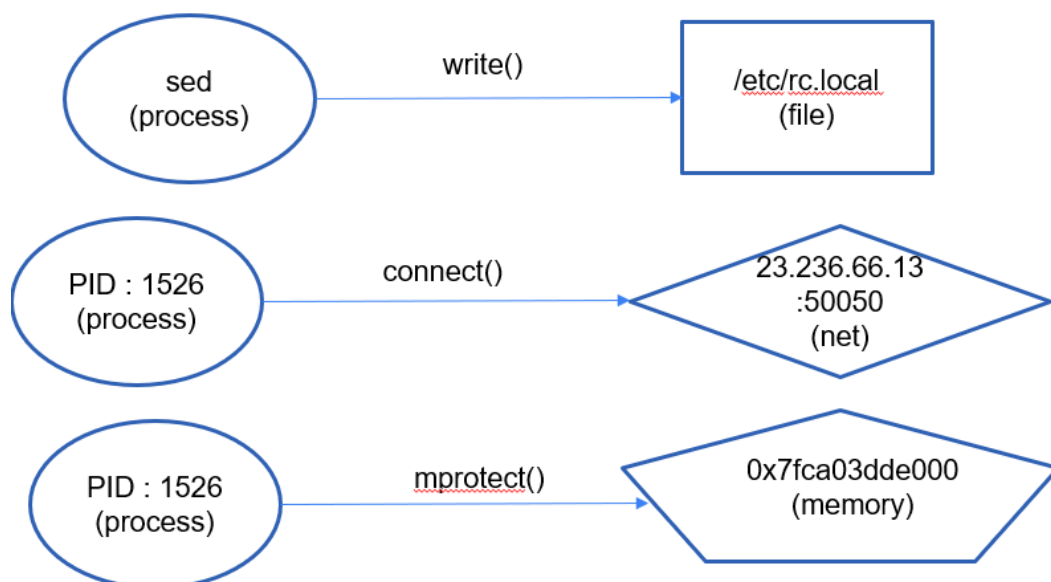


Figure 2.1: ASG Structure Example

2.1.3 Algorithm



本論文建構 Attack Scenario Graph (ASG) 的演算法如 Algorithm.1 所示，我們將利用系統工具”strace”動態分析取得的數個 System Call Trace 當作 Input 用來產生 Attack Scenario Graph，會有數個 System Call Trace 是因為一個惡意程式可能有數個子 process，每一個 process 都會有一個對應的 system call trace。針對 trace 中的每一行，本論文會利用 SystemCallParser 模組進行處理，如同章節 2.1.1 所介紹 system call trace 的範例，我們將從 trace 中的每一行萃取出 3 種資訊，分別是 1. system call name、2. parameters、3.return value，並且會進一步地從 parameters 中找出我們關注的作用對象。SystemCallParser 首先會使用 regular expression 的方式萃取出其 system call name、parameters、return value，因為每個 System Call 的參數量及關注的受詞位置可能不同，我們需針對每一個 system call 單獨處理取得其受詞，以圖片中的範例為例，read() 這一 system call 我們關注的受詞是其讀取的檔案描述符，位於第 1 個參數、mmap2() 這一個 system call 關注的是他將哪一個檔案放進記憶體中使用，一樣需要找到他的檔案描述符號，而這次則是位於第 5 個參數，這些資訊皆可以在參考資料 [3] Linux man page 取得，我們將 Linux 版本的 405 個 system call 逐一檢視，挑選出所關切的最重要的一個受詞物件參數。接著我們會透過 connect_node() 的函數將取得的 system call name 和 object 的資訊作為 Edge、Node 存入 Graph 資料結構中，待整個 Graph 建立完成，會使用 Visualizer() 函數將整個 Attack Scenario Graph (ASG) 視覺化。

Algorithm 1 ASG Algorithm



```
SystemCallParser(line):
    info = re.search(' (.*?)(.*).*= (.*?)', line) # using regex to parse the line
    syscall_name = info[0], parameters = info[1], return_value = info[2]
    if syscall_name == read then
        object = parameters[0]
    else if syscall_name == mmap2 then
        object = parameters[4]
        .
        .
        .
        # The object may in different position, use the pattern to solve #
    end if
    return syscall_name, object, return_value

ASG(traces):
    Graph = # Store Node and Edge connection status
    for trace in traces do
        for line in trace do
            syscall_name, object, return_value = SystemCallParser(line) # parse the line
            connect_node(Graph, syscall_name, object) # connect the node and edge into the
            Graph
        end for
    end for
    Visualizer(Graph)
```



2.2 Related Work

2.2.1 Dynamic Malware Analysis in the Modern Era—A State of the Art Survey

論文 [6] 分析了為何現今惡意程式的分析大多是使用所謂的動態分析 (Dynamic Malware Analysis) 的方式來分析一惡意程式。文中有提到一個很重要的觀點，惡意程式的撰寫者能透過各種方式的規避靜態分析，例如：代碼混淆、動態代碼加載、加密和打包，使研究人員難以從靜態分析中找到惡意行為，但這些行為透過動態分析的方式便能夠側錄下來，因此作者認為動態分析 (Dynamic Malware Analysis) 在分析惡意程式上是一個較好的做法。本論文為了能盡量完整的取得惡意程式的行為，因此也會採用動態分析的方式來側錄惡意程式，取得其 System Call Trace 並進一步的分析。

2.2.2 HOLMES: Provenance Graph 與 Audit logs

論文 [5] 提出了一個 HOLMES System，目的是為了能夠即時的偵測 APT 活動 (Real-time ongoing APT Campaign detection)，主要的作法為根據 Audit log 產生一個高階的 graph 來總結目前偵測到的 APT 活動，方便研究人員依照此 Graph 進行對應。

此論文針對 Graph Node 的定義共有 5 種類別，分別是 1.File，例如 /etc/passwd、/usr/local/bin/sed 這種存於電腦中的檔案、2.Process，例如執行 cat 指令產生的 process、執行 bash 指令產生的 process、3.Network，例如一個 socket IP、4.Memory，例如一記憶體位置，但論文中並非真實將記憶體位置呈現，而是以一個 Memory Node 表示、5.User，例如是普通權限的 user 或是 root 權限的 user。

Graph Edge 的部分則是代表 System Call，像是使用 `exec()` 執行一個指令，或是使用 `read()` 讀取一個檔案的內容。

與本論文相似的 Node 設計有其前 4 個種類，第 5 個 user 種類經過我們實際側錄惡意程式後發現其實包含於我們所設計的 ID 類別中，像是惡意程式使用 `getuid()` 如果取得 `id=0` 就可得知是 root 權限，並且 `id` 也可以取得更豐富的權限資訊，像是 `pid` (process 權限)、`gid` (group 權限)，因此我們認為 ID 類別是較佳的選擇。Edge 的部分與本論文所欲設計的 Edge 一致，都是使用 System Call 來代表惡意程式執行的動作，

2.2.3 Tactical Provenance Analysis for Endpoint Detection and Response Systems : Provenance Graph 與 API call logs

論文 [1] 提出了一個方法，將 Endpoint Detection and Response(EDR) 所搜集的資料，以 Provenance Graph 的形式呈現，方便資安人員進行分析，用以解決 EDR 資料龐大、低階、難以分析的問題。

此論文會將 EDR 搜集的資料以 Provenance Graph 的方式進行呈現，其 EDR 的資料為 Window API Call Trace，與本論文所研究的 System Call Trace 同屬一種 Execution Trace 的表達方式，其呈現的 Graph Node 共有 2 個類別，1. Process 類別，例如啟動一個 terminal 所相關的 process node，對應到 Linux 系統中類似於 `sh`、`bash` 等指令所產生的 process、2. Object 類別，論文中未對其做完整的定義，只有告知我們其中可以包含如 File、Socket、Window registry 等物件，論文中亦只看到這些範例，可能是保留一個空間，待分析更多案例後能夠增加新類別。Graph Edge 的部分則是直接以 API Call 表示。

與本論文相似的 Node 設計中除去 Window registry 的部分，因為本論文分析

的惡意程式皆屬於 Linux 平台，並不會有 registry 相關的資訊。Edge 的部分本論文會以 System Call 來代表，與 API Call 同樣是用來代表惡意程式所執行的動作。



2.2.4 AttackKG: Constructing Technique Knowledge Graph from Cyber Threat Intelligence Reports

論文 [4] 觀察到目前安全從業者以網絡威脅情報 (CTI) 報告的形式積極總結和交流有關跨組織攻擊的知識。然而，由於以自然語言文本編寫的 CTI 報告不適合自動分析，因此提出一個方式將這些 CTI 報告轉換成一個結構化 Graph。

此論文中對 Graph Node 的定義可以分為 6 個類別。1.File 類型，例如一個.doc 檔案、/etc/passwd 檔案、2.Executable 類型，例如一可執行的腳本、3.Network 類型，例如一 socket IP，或是直接以網址呈現、4.Registry，例如一 Windows registry、5.Actor，因為此論文的資論來源是 CTI Report，研究人員也會關注 Report 中的 actor 是誰，因此有此一類別、6.Other 類別，論文有提到有些物件難以對應到一個類別，像是 email、CVE 等等，因此他們將這些歸類於 other 類別。針對 Graph Edge 的部分論文只有定義它用來代表 Node 之間的”dependencies”，所以在 AttackKG 論文的 Graph 中 Edge 上面不會有任何的東西。

與本論文相似的點 (node) 為其中的前 3 個類別，分別是 1.File、2.Network、3.Executable。Executable 類別其實就類似於前述幾篇論文的 Process 類別，因為在 AttackKG 的論文中我們可以看到它會用來代表一可執行的腳本，其執行後對應的就是一個 process，類似於在 Linux 平台執行 sh 產生對應的 process，因此我們的類別命名會以 Process 表示。Registry 類別不取用的原因如前述所言，我們的實驗專注於 Linux 平台，因此不會有 Registry 的資訊，Actor 類型則是因為一個惡意程式中的側錄結果並不會有 actor 的資訊，這是一篇 CTI Report 中才會有的資訊，因

此不參考。Graph Edge 的部分此論文中只有表示”dependencies”，我們認為 Edge 上帶有 System Call 能讓 Graph 的資訊量更豐富。





Chapter 3 Automated Narrative Essay Generation

3.1 System Architect

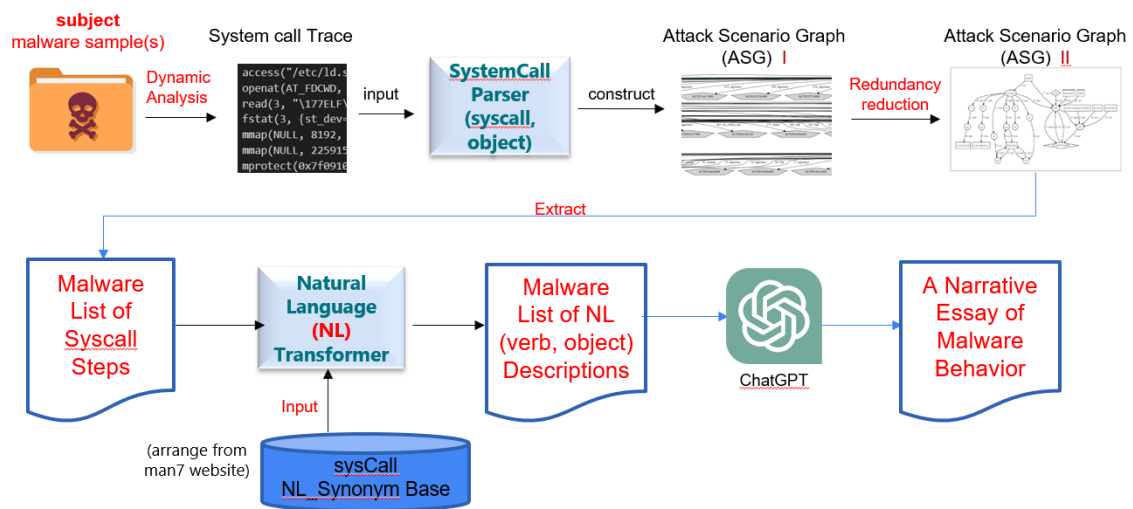


Figure 3.1: System Arch Automatic Narrative Essay Generation

Figure3.1呈現本論文的系統架構。整個運做的流程為首先取得欲分析的惡意程式的樣本，利用系統工具”strace”透過動態分析 (Dynamic Analysis) 的方式獲取其 System Call Trace，接著將此 System Call Trace 放進 SystemCallParser 中取得 system call name 及 object (selected parameter value), 我們會依此資訊繪製 Attack Scenario Graph I (ASG I)，其中 system call name 會作為 Edge、而 Obejct 會作為

Node 呈現，由於 System Call Trace 的數量通常非常龐大，因此 ASG I 常常會難以以視覺化方式呈現，罔論清楚辨識其作業。因此，我們提出 Redundancy reduction 的方式來降低其大小規模，得到 Attack Scenario Graph II (ASG II)。

本論文將從 ASG II 中存取每一個詳細步驟並將其存於一資料結構中 (Malware List of Syscall Steps)，因為 ASG II 是依據 System Call Trace 所繪製，所以每一個詳細步驟的內容皆為低階語言，因此我們設計一個轉換模組 Natural Language Transformer，其功能為透過 Natural Language Synonym Base 的輔助，將每一個詳細步驟轉換為自然語言描述，最後將此自然語言描述放進 ChatGPT 中，透過 ChatGPT 的輔助生成惡意程式的自然語言敘事文本。

3.2 Issue: ASG Large Scale

一般而言，惡意程式的 system call trace 相當地詳細且數量龐大，可達數千、數萬甚至數十萬不等，例如：對同一個檔案進行數百次的 open()、read()、write()、生成一個新的子程序連帶產生數次的記憶體位置操作、或是不定期使用 time() 這一系統呼叫來取得當前時間戳記。這將使我們產生出的 ASG 難以閱讀，因此我們提出四個 Reduction 方式，縮減的部份會保留其完整資訊於一資料結構中，並不會有資訊的遺失。

3.2.1 For Node

根據我們的觀察，每生成一個新的子程序就會連帶產生多次的記憶體位置操作，例如使用 system call brk()，來設定新的子程序的記憶體斷點，或是使用 system call lock() 將子程序所要存取的記憶體位置鎖起來，避免發生同步的問題，這導致 Graph 中有大量記憶體位置相關的 Node；另外惡意程式在其整個生命週期

中也常常使用 system call time() 來取得當前時間戳記，這會導致時間相關的 Node 數量非常龐大；最後我們也觀察到惡意程式會嘗試取得所有 Network Interface 的資訊，例如使用 system call read() 讀取”eth0、eth1、eth2...”這些網路介面卡的資訊，因此 Graph 中也會有數個 NIC 相關的 Node。

統整以上觀察，我們得知 ASG 中有大量對 1. 記憶體、2. 當前時間查詢、3. NIC 的操作，導致 Node 數量過於龐大而不利視覺辨識，我們預計將這些綜整，只在 Graph 中呈現一次，規則如下：

- create a “memory” sink node: 我們不在意對哪一塊記憶體位址做操作，只記錄有對記憶體操作的事實，所以將不同的 memory addresses，以一個 sink node 呈現，該 node 會有一資料結構紀錄詳細的記憶體位址（資訊不遺失）
- create a “timestamp” sink node: 我們不在意惡意程式用 time() 來取得多個不同的當前時間，所以以一個 sink node 呈現，該 node 會有一資料結構紀錄詳細的時間資料（資訊不遺失）
- create a “NIC” sink node: 我們不在意是從哪個 eth 中取得取得資訊，因此以一個 sink node 呈現，該 node 會有一資料結構紀錄詳細的網路介面資料（資訊不遺失）

3.2.2 For Edge

根據我們的觀察，惡意程式對同一個檔案時常會進行多次相同的操作，例如針對同一個檔案進行數千次的 read()，這會使 Graph 中有數千條同樣代表 read() 的 Edge 指向同一個檔案，導致 Graph 過於龐大而不利視覺辨識。針對這個現象我們預計將其 Edge 統整成一條，用以保留其操作之事實，但仍會有一資料結構保存期完整的資訊。



3.2.3 Reduction Result

經過上述處理，我們可以得到 Reduction 後的 ASG (ASG II)，Steps 的部分會有一資料結構完整保留因此數量不變，Node 以及 Edge 則會有所下降，我們以 Malware ”Dofloo” 為案例，並將詳細變化紀錄於 Table 3.1: Dofloo Scale Reduction。ASG 的變化紀錄於 Fig. 3.2 3.3 3.4 3.5。

Table 3.1: Dofloo: Scale Reduction

Dofloo	ASG I(original)	ASG II(Reduced)
# of steps	7577	7577
# of source nodes	11	11
# of destination nodes	392	54
# of edges	7577	284

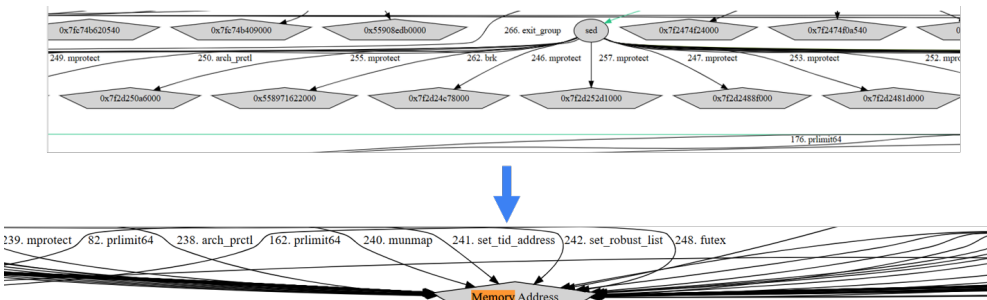


Figure 3.2: Memory Reduction

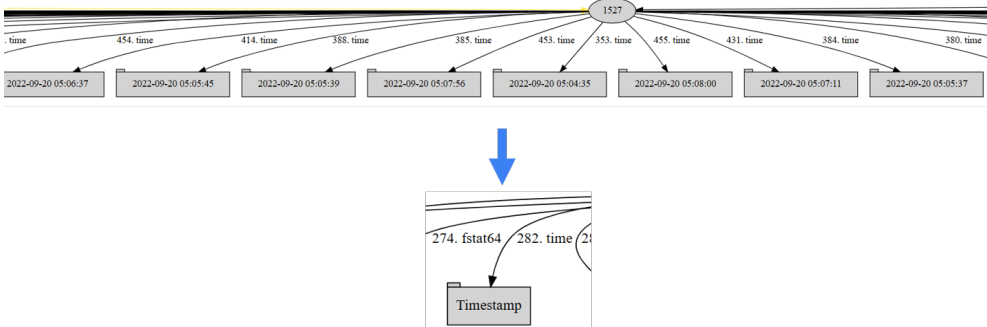


Figure 3.3: Time Reduction

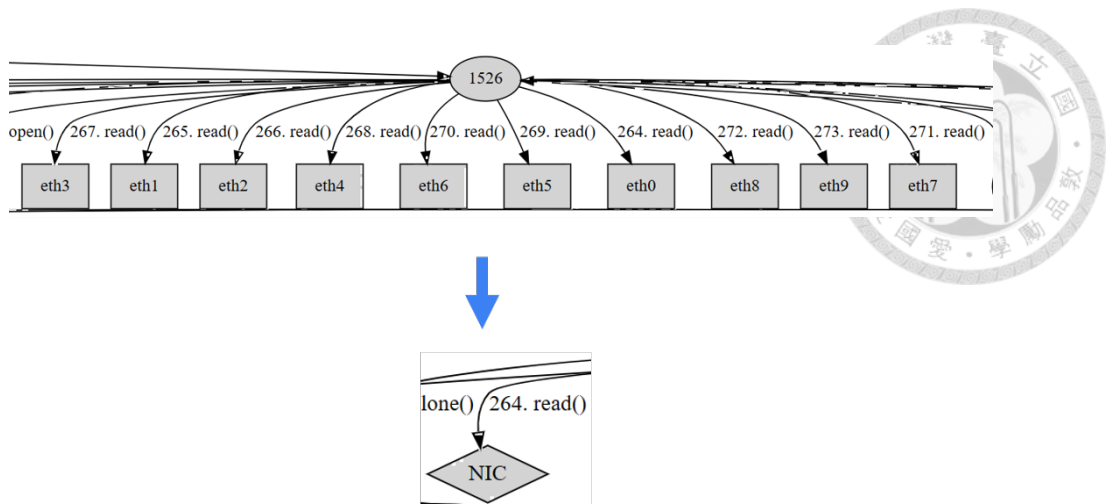


Figure 3.4: NIC Reduction

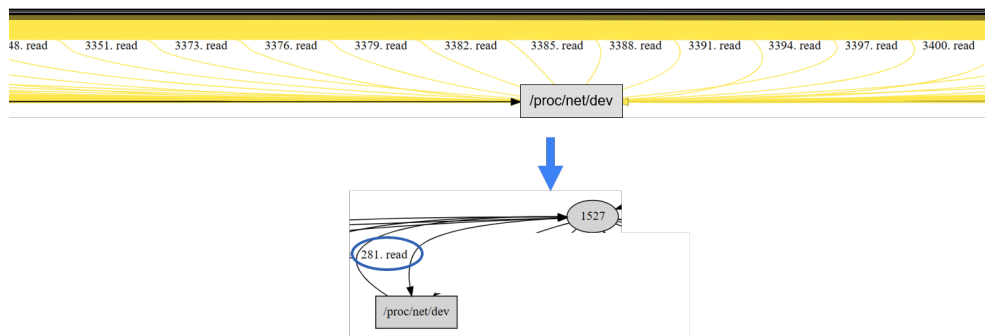


Figure 3.5: Steps Reduction

依此 Graph Reduction 的規則，我們同樣對 Xorddos、Darlloz、LuaBot、Kaiten 等 Family 進行實驗，結果呈現於 table.3.2 3.3 3.4 3.5

Table 3.2: Xorddos: Scale Reduction

Xorddos	ASG I(original)	ASG II(Reduced)
# of steps	112417	112417
# of source nodes	667	667
# of destination nodes	4214	814
# of edges	112417	7469

Table 3.3: Darlloz: Scale Reduction

Darlloz	ASG I(original)	ASG II(Reduced)
# of steps	2075	2075
# of source nodes	27	27
# of destination nodes	524	315
# of edges	2075	596

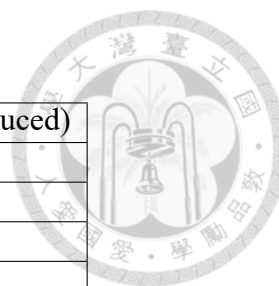


Table 3.4: LuaBot: Scale Reduction

LuaBot	ASG I(original)	ASG II(Reduced)
# of steps	9746	9746
# of source nodes	3	2
# of destination nodes	700	288
# of edges	9746	349

Table 3.5: Kaiten: Scale Reduction

Kaiten	ASG I(original)	ASG II(Reduced)
# of steps	344	344
# of source nodes	2	2
# of destination nodes	42	16
# of edges	344	38

3.3 Issue: System call 的低階結構、語義

因為 system call 的低階程式結構與語義，研究人員常常難以快速掌握其中的重要行為或執行企圖，需要花費大量時間去進行解讀。本論文認為如果能將其轉化為自然語言，並且能夠將執行步驟做綜整並重點提示，將會對要瞭解惡意程式的人士提供一個非常實用的工具。

3.4 Solution

本論文預計先將 ASG 中的各個 Steps 轉換成自然語言，並以此自然語言為基礎，透過 ChatGPT 的輔助產生一篇易於人類閱讀的文章，舉例來說: Fig.3.6 中 malware process 執行 clone() system call 產生 sh，我們先將其轉換為一段自然語言描述:「malware」「create」a child process : sh，這個轉換我們建構了一個稱為 NL_Synonym_Base 知識庫的輔助進行動詞、受詞描述的轉換。透過這種模式我們能將整個 ASG 的每一個步驟都轉化為自然語言描述，最後再經由 ChatGPT 的協助產生整個惡意程式執行活動的敘事文本。

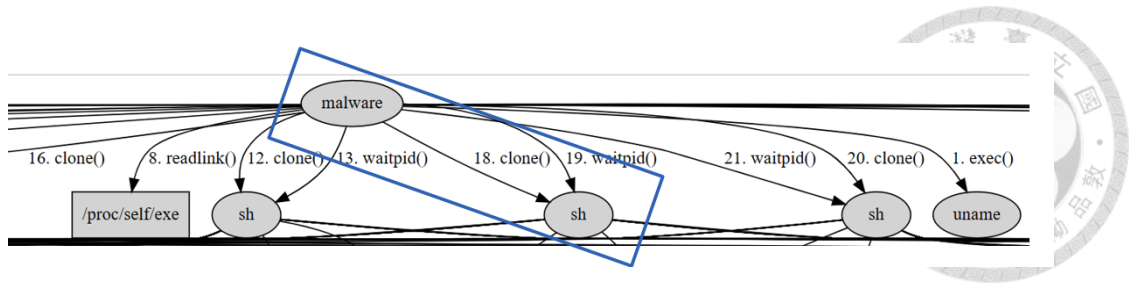


Figure 3.6: ASG Step Example

3.4.1 Malware List of Syscall Steps

Malware List of Syscall Steps 是 ASG 中的一個資料結構，用來儲存 ASG 中所有 Steps 的資訊，如 Fig. 3.7。這個資料結構即可代表一個惡意程式在執行週期中各種活動，例如圖中的第一個步驟我們可以得知惡意程式使用 System Call ”exec()”，執行了 uname 這個指令、第二個步驟則是針對記憶體位置”0x8fe7000”執行了 brk() 這個 System Call。我們預計將這其中的 System Call 轉換為一個動詞片語描述，並將其作用的物件轉換為一受詞片語描述，這一動作會經由一稱為 NL_Synonym Base 的輔助，透過 Natural Language Transformer 來完成轉換。

Step 1 : malware → exec() → uname
 Step 2 : malware → brk() → 0x8fe7000
 Step 3 : malware → set_thread_area() → 0x8fe7860
 Step 4 : malware → set_tid_address() → 0x8fe78c8
 Step 5 : malware → set_robust_list() → 0x8fe78d0
 Step 6 : malware → futex() → 0xffe57718
 Step 7 : malware → ugetrlimit() → 8192*1024 bytes
 Step 8 : malware → readlink() → /proc/self/exe
 Step 9 : malware → getcwd() → /prober
 Step 10 : malware → clone() → sh
 Step 11 : malware → waitpid() → sh

Figure 3.7: ASG Steps List Example



3.4.2 NL_Synonym Base

如同前述所言，我們希望將 ASG 中的各個 Steps 的 System Call 轉換為一個動詞片語描述，並將其作用的物件轉換為一受詞片語描述，我們參考 Linux man page [2] 的 Name and Description sections 對於一 system call 的語義描述，擷取該 system call 描述中的動詞片語 (verb phrase) 以及作用受詞描述片語，如 Fig. 3.8。透過其輔助我們可以將 system call 轉化為對應的動詞、受詞描述，例如: System call “clone()”，其對應的動詞片語為 “create”，其對應的作用受詞描述片語為 “a child process”，因此 malware -> clone() -> sh 會依照規則轉化為”malware” create a child process : sh。

`clone, __clone2, clone3 - create a child process`

Figure 3.8: Linux man page example

3.4.3 schema

NL_Synonym Base 的 schema 如 table.3.6 所示，主要包含 3 個欄位。1.System Call Name: 用來代表轉譯之前原始的 system call function name、2. 動詞片語: 用來代表轉譯後的動詞、3. 作用受詞描述片語: 用來代表轉譯後的受詞描述。如同章節 3.4.2 的範例，我們可以依此 schema 將 system call 轉譯成對應的動詞片語描述及受詞片語描述。例如針對 clone() 此一 system call，我們可以轉譯成 create a child process 的自然語言描述、針對 exit() 則是可以轉譯為 terminate the calling process。使用片語一詞是因為動詞及受詞不一定單單是一個詞語，像是 table.3.6 的最後一個案例中，wait4() 轉譯後的動詞會是 “wait for”，此為一動詞片語描述；而表中的 brk() 轉譯後的受詞會是 “the location of the program break”，也並非單單是一個受詞，因此本論文會以片語的形式來描述。

Table 3.6: NL Synonym base schema

System Call Name	動詞片語	作用受詞描述片語
clone()	create	a child process
exit()	terminate	the calling process
linkat()	create	a file link
rename()	change	name or location
brk()	change	the location of the program break
open()	open	a file or device
getpid()	get	process identification
exec()	execute	a program
wait4()	wait for	a process

3.4.4 Natural Language (NL) Transformer

Natural Language Transformer 是本論文中真正執行轉換動作的功能模組，其功能為將 ASG graph 裡的每一個 system call invocation (step) 轉譯為自然語言的描述，模組的輸入及輸出如下所示：

- Input: 1. Malware List of Syscall Steps (verb, object)、2. NL_Synonym Base
- Output: 1. Malware List of Primitive NL Descriptions

透過這個模組我們可以將 ASG 中的各個 Steps 轉皆轉化為一簡易 (primitive) 自然語言描述，如 Fig. 3.9 所示。例如: Step 1 : malware -> exec() -> uname 被轉譯為 Step 1 : "malware" execute program : uname、Step 2 : malware -> brk() -> 0x8fe7000 被轉譯為 Step 2 : "malware" change the location of the program break : 0x8fe7000。此等轉譯已能讓研究人員更容易理解惡意程式的執行行為，例如尚未轉譯之前的 Step 2: malware -> brk() -> 0x8fe7000，研究人員可能無法得知 brk() 到底是對

記憶體做了甚麼操作，但是轉譯後的 Step 2: "malware" change the location of the program break: 0x8fe7000 就能讓研究人員清楚的得知，惡意程式在這個步驟是在設定程式的斷點。



然而這個階段仍是將惡意程式的每個步驟一一列出，從 Fig. 3.9 中我們可以觀察到，Step 2: "malware" change the location of the program break: 0x8fe7000 和 Step 3: "malware" set thread local storage area: 0x8fe7860，這兩個步驟皆是在做記憶體位置的操作，我們認為若能將這種有關聯性的步驟綜整成一個一個的段落，並更進一步的產生一篇文章，將能為研究人員提供更容易閱讀與理解且包含背景與語義豐富的資訊。因此我們嘗試將此轉譯過後的惡意程式執行步驟，作為 ChatGPT 的 Input，透過其輔助自動化的產生一篇惡意程式的敘事文本。

3.4.4.1 Algorithm

此模組的演算法如 Algorithm.2所示，演算法如同前述所言，共有兩個 Input，分別為 1. Malware List of Syscall Steps (verb, object)、2. NL_Synonym Base，我們會透過遍歷所有的步驟來完成簡易自然語言的轉換。首先，針對每一個步驟我們會取得其 3 個資訊，分別是 1.sourceNode: 代表執行 System Call 的主體、2.syscall: 代表其執行的動作、3.destNode: 代表其作用的受體。接著我們透過 synonymBase 的輔助，根據其 syscall，找到對應的動詞、受詞片語，這個過程如同章節 "NL_SynonymBase" 中提及的 schema 3.6所示，如果是 system call clone() 我們將會找到動詞片語 "create"、受詞片語 "a child process"；如果是 system call exit() 就會找到動詞片語 "terminate"、受詞片語 "the calling process"。最終我們將這個轉化後的動詞、受詞片語，與原始的節點資訊串聯起來，存於一資料結構 NL_Step 中，並且將此結構依序存入 NL_stepList 中，用以儲存所有轉換後的簡易自然語言步驟，最後將其返回給使用者。



Algorithm 2 NL Transformer Algorithm

```
NL_Transformer(stepList, synonymBase):  
    NL_stepList = [] # store the result  
  
    for step in stepList do  
        # get the step info  
        sourceNode = step["sourceNode"]  
        syscall = step["edge"]  
        destNode = step["destNode"]  
  
        # use synonym base to get the verb and object  
        verb = synonymBase[syscall]["verb"]  
        object = synonymBase[syscall]["object"]  
  
        # store the result  
        NL_step = sourceNode + verb + object + " : " + destNode  
        NL_stepList.append(NL_step)  
    end for  
    return NL_stepList
```

Step 1 : malware → exec() → uname

Step 1 : "malware" execute program : uname

Step 2 : malware -> brk() -> Memory Address

Step 2 : "malware" change the location of the program break : 0x8fe7000

Step 3 : malware -> set_thread_area() -> Memory Address

Step 3 : "malware" set thread local storage area : 0x8fe7860

⋮

Step 10 : malware -> clone() -> sh

Step 10 : "malware" create child process : sh

Figure 3.9: Natural Language (NL) Transformer Example



3.5 案例分析: Dofloo

本論文以 Dofloo 為例，嘗試將 Natural Language Transformer Example 轉換後的惡意程式執行步驟，透過 ChatGPT 的輔助來產生敘事文本，結果如 Fig. 3.10 3.11 3.12所示。從圖中我們可以觀察到 ChatGPT 主要進行兩項任務: 1. 將有關聯性的步驟綜整成一個一個段落，並以正規的英文形式撰寫，例如: Fig. 3.10 中的 Step 2 和 Step 3 如前一個段落所言，都是針對記憶體位置的操作，圖中就可以看到 ChatGPT 將其綜整為一個段落”Memory Manipulation and Control”，並以正規英文形式撰寫。2. 對專有技術名詞補充說明，例如 Fig. 3.10 中的 Step 1: ”malware” execute program : uname，我們直接看這一個步驟只知道惡意程式執行了 uname 這一程序，但是無法得知 uname 有何作用，從圖中就可以看到 ChatGPT 對 uname 這種技術名詞進行補充說明，告訴我們 uname 這個指令可以用來取得 kernel version。Dofloo 案例中共有 284 個 Steps，完整實驗結果會於參考資料附上 [7]。

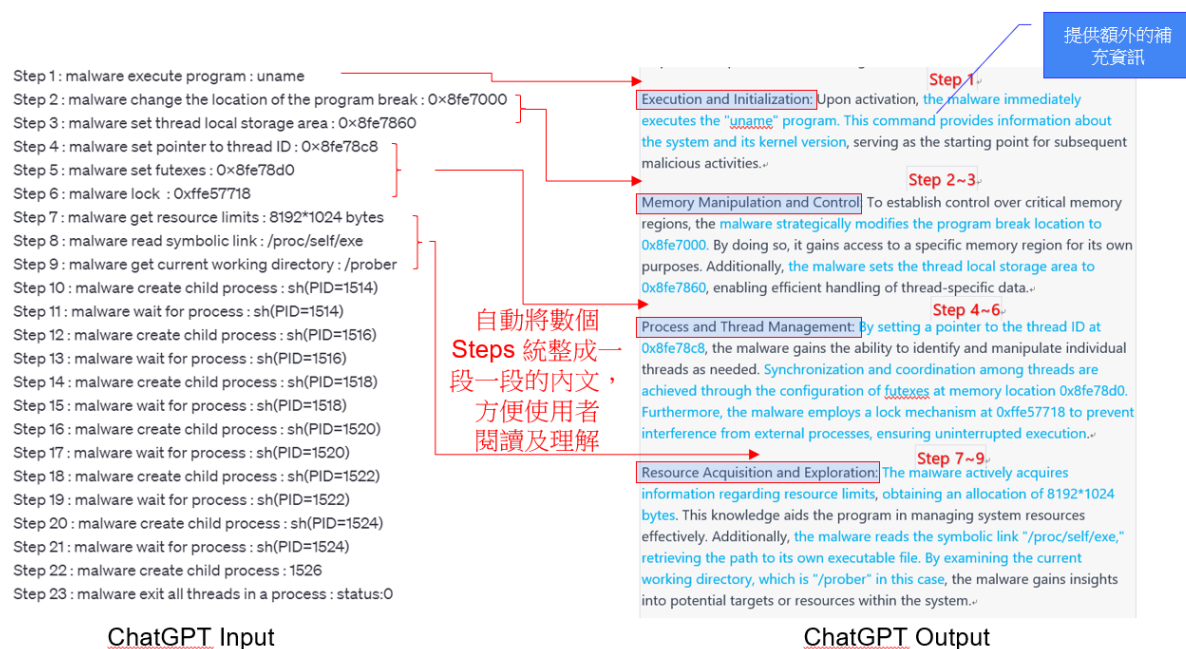


Figure 3.10: ChatGPT Dofloo Example 1

Step 1: malware execute program : uname
 Step 2: malware change the location of the program break : 0x8fe7000
 Step 3: malware set thread local storage area : 0x8fe7860
 Step 4: malware set pointer to thread ID : 0x8fe78c8
 Step 5: malware set futexes : 0x8fe78d0
 Step 6: malware lock : 0xffe57718
 Step 7: malware get resource limits : 8192*1024 bytes
 Step 8: malware read symbolic link : /proc/self/exe
 Step 9: malware get current working directory : /prober
 Step 10: malware create child process : sh(PID=1514)
 Step 11: malware wait for process : sh(PID=1514)
 Step 12: malware create child process : sh(PID=1516)
 Step 13: malware wait for process : sh(PID=1516)
 Step 14: malware create child process : sh(PID=1518)
 Step 15: malware wait for process : sh(PID=1518)
 Step 16: malware create child process : sh(PID=1520)
 Step 17: malware wait for process : sh(PID=1520)
 Step 18: malware create child process : sh(PID=1522)
 Step 19: malware wait for process : sh(PID=1522)
 Step 20: malware create child process : sh(PID=1524)
 Step 21: malware wait for process : sh(PID=1524)
 Step 22: malware create child process : 1526
 Step 23: malware exit all threads in a process : status:0

相似的 Steps 會統
 整成一段話來描述

Step 10~23

Child Process Creation and Control: As part of its operations, the malware creates multiple child processes, each spawned using the "sh" command. These child processes are sequentially created and managed. The malware waits for the completion of each process before proceeding to the next one, ensuring an orderly progression. The creation of these child processes, including the final process with PID 1526, serves the malware's specific objectives.

ChatGPT Input

ChatGPT Output

Figure 3.11: ChatGPT Dofloo Example 2

Step 24: "sh(PID=1514)" change the location of the program break : 0x55cafcdb000
 Step 25: "sh(PID=1514)" set region of memory : 0x7fca03dde000
 Step 26: "sh(PID=1514)" set thread state : 0x7fca041f5540
 Step 27: "sh(PID=1514)" unmap files or devices : 0x7fca041f6000
 Step 28: "sh(PID=1514)" get user identity : UID:0
 Step 29: "sh(PID=1514)" get group identity : GID:0
 Step 30: "sh(PID=1514)" get process identification : PID:1514
 Step 31: "sh(PID=1514)" get user identity : UID:0
 Step 32: "sh(PID=1514)" get process identification : PID:1513
 Step 33: "sh(PID=1514)" get file status : /prober
 Step 34: "sh(PID=1514)" get file status :
 Step 35: "sh(PID=1514)" get group identity : GID:0
 Step 36: "sh(PID=1514)" get file status : /usr/local/sbin/sed
 Step 37: "sh(PID=1514)" get file status : /usr/local/bin/sed
 Step 38: "sh(PID=1514)" get file status : /usr/sbin/sed
 Step 39: "sh(PID=1514)" get file status : /usr/bin/sed
 Step 40: "sh(PID=1514)" get file status : /sbin/sed
 Step 41: "sh(PID=1514)" get file status : /bin/sed
 Step 42: "sh(PID=1514)" execute program : sed(PID=1515)
 Step 43: "sh(PID=1514)" wait for process : PID:-1
 Step 44: "sh(PID=1514)" exit all threads in a process : status:0

Establishing Control over Critical System Resources: The malware's first objective is to establish control over critical aspects of the system's memory and thread management. By modifying the program break location, it skillfully gains access to a specific memory region (0x55cafcdb000). This alteration serves as the foundation for subsequent operations.

Step 24

Efficient Thread and Memory Management: To ensure efficient thread and memory management, the malware configures the thread local storage area (0x7fca03dde000) and sets the thread state (0x7fca041f5540). It unmaps files or devices at memory location (0x7fca041f6000). This action allows the malware to detach and free up resources, contributing to its efficient resource management strategy. These measures allow for seamless handling of thread-specific data, enhancing the malware's ability to execute its malicious intent.

Step 25~27

Gathering Essential System Information: To navigate the system effectively, the malware collects essential information about the user identity, group identity, and process identification. Armed with this data (UID:0, GID:0, PID:1513, 1514), the malware solidifies its privileged position within the system.

Step 28~32

Analyzing File Statuses for System Exploration: Further exploration involves inspecting various file statuses. The malware scrutinizes the "/prober" directory, the current directory, and specific paths associated with the "sed" program. These evaluations empower the malware to gain insights into the system's file structure and identify potential targets for manipulation.

Step 33~34

Step 36~41

能夠從 Steps 中統整
 出惡意程式在存取
 sed 相關的檔案路徑

ChatGPT Input

ChatGPT Output

Figure 3.12: ChatGPT Dofloo Example 3

3.6 Issue: 自動化標註

透過上述的實驗，我們已經可以將一個惡意程式透過 ChatGPT 以自然語言的方式描述，並且透過人工的方式我們也請資安專家閱讀產出的文本確認了 ASG 中的各個 Steps 皆有在文章中被提及。在此確認過程中，專家必須以人工方式逐一對所產出的敘述段落對應 ASG 的步驟，相當耗時且費力，因此我們希望能讓

ChatGPT 在產生文章時一併將對應的 Steps 標示清楚，降低人力的耗費。



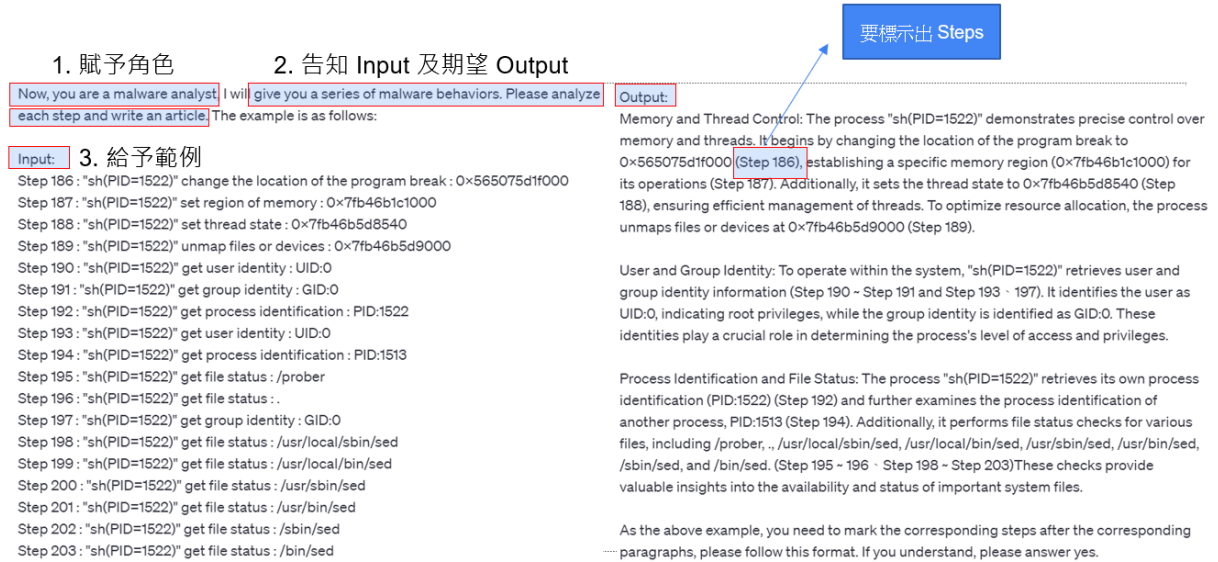
3.7 Solution: ChatGPT Prompt

ChatGPT Prompt 是一種使用 ChatGPT 模型的技巧，用以引導 ChatGPT 產出符合自己需求的內容。本論文希望透過研究 ChatGPT Prompt 來讓其能夠自動化標示對應的 Steps。

本論文的 Prompt 主要包含 4 點：1. 賦予角色、2. 告知 Input 及期望的 Output、3. 給予範例、4. 給予 Step List。前 3 點主要用於建立使用情境，第 4 點則是我們真正的輸入。

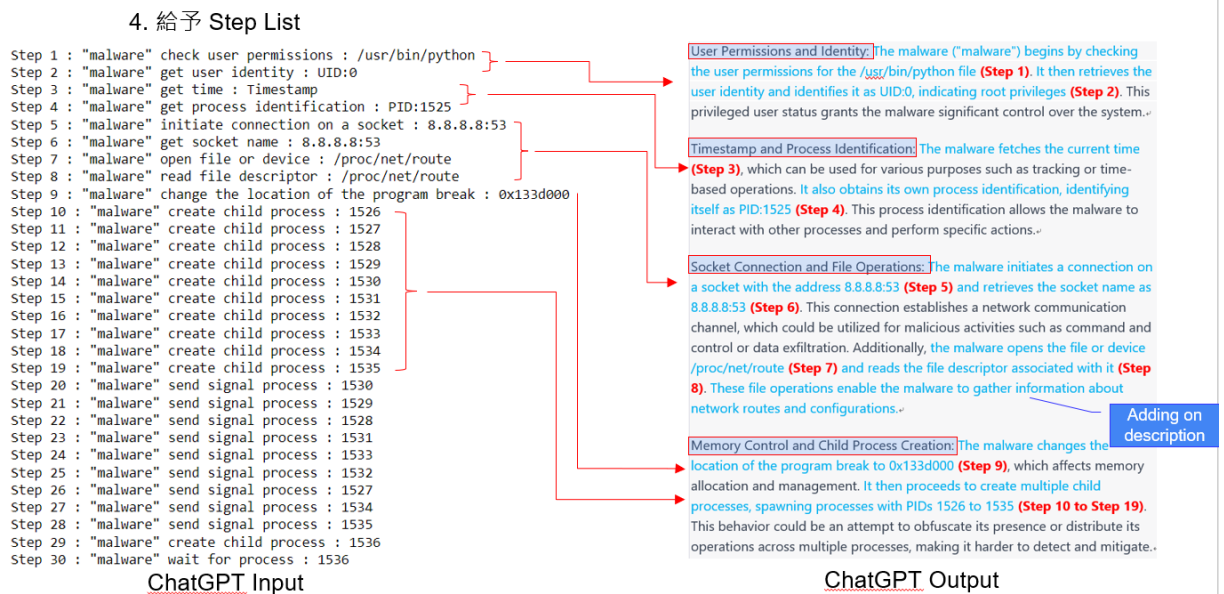
舉例來說，如 Fig. 3.13 3.14 所示：1. Prompt 的一開始我們會輸入 "Now, you are a malware analyst"，讓 ChatGPT 了解現在它將扮演一個惡意程式分析師的角色；2. 接下來我們會輸入 Prompt "I will give you a series of malware behaviors. Please analyze each step and write an article"，告知我們預計給予一系列的惡意程式行為，請它幫我們分析並綜整成一篇文章；3. 純粹這樣做其實無法讓 ChatGPT 為我們標示出每一個 Step 對應在文章中的哪一個句子，會得到如同 Fig.3.10 所示，對應句子沒有標註對應 Step 的輸出，因此我們必須先訓練 ChatGPT。我們的做法是將前一階段（例如 Dofloo 案例分析中的 Input 及 Output）當作一個新的 Prompt，並且在 Output 的地方以人工標註上對應的 Step。如 Fig.3.13 所示，我們告知 ChatGPT 期望的 Input 是經由 Natural Language Transformer 轉譯後的惡意程式執行步驟（圖片左半部中 "3. 給予範例 Input"），而期望的 output 是一篇正規英文文章的形式，並且要標註對應的 Steps（圖片右半部中 "Output"），這個訓練過程中我們人工標示了 Step 186 - Step 203 共 17 個 Steps，如 Fig.3.13 左半部所示，並不需要重複給予非常多的範例，透過這樣的方式 ChatGPT 就能清楚的了解我們期望的 Input 及 Output

格式。4. 最後我們只要如 Fig.3.14的左半邊所示，給予一個經由 Natural Language Transformer 轉譯後的惡意程式執行步驟，ChatGPT 就會產生圖中右半邊的自然語言敘事文本，並且為我們一一標記上各執行步驟所對應的 Steps。



ChatGPT Input

Figure 3.13: ChatGPT Prompt 1



ChatGPT Input

ChatGPT Output

Figure 3.14: ChatGPT Prompt 2



3.7.1 Discussion

3.7.1.1 Version

本論文所使用的 ChatGPT 版本為 3.5，這個版本能夠免費使用並提供網頁介面進行操作，不選用 ChatGPT API 的原因是 API 免費版本為 3.0，並且有使用限制，一定數量的 token 後就會無法繼續使用 API。

3.7.1.2 Limitation

在進行實驗的當下 ChatGPT 官方釋出的 Prompt 可用 Token 數為 1024，這代表每次輸入的 Prompt 大小如果超過 1024 就有可能會被忽略，本論文的實驗經驗每次輸入的 Steps 盡量維持在 30 個以下會有較好的成效。

3.7.2 Evaluation Output

ChatGPT 屬於生成式模型，因此產生的 Output 可能會有錯誤，我們將針對這部分進行評估。

根據我們的實驗結果，ChatGPT 綜整的文章基本上沒有任何錯誤，嘗試分析原因後本論文認為這是因為我們的 Prompt 品質良好，如 Fig. 3.10 所示，可以看出我們在給予 ChatGPT Input 時已經透過 Natural Language Transformer 的協助，將每一個 Steps 都轉化為一個簡易自然語言的句子，例如，圖中的 Step 2、Step 3 分別為”malware change the location of the program break : 0x8fe7000”、”malware set thread local storage area : 0x8fe7860”，已是自然語言描述的句子；並且 Prompt 中給予 ChatGPT 的任務也非常具體，就是依據這些已經是自然語言的句子，分析並綜整成一篇文章，在 Fig. 3.10中能看見 ChatGPT 將 Step 2 和 Steps 3 整合成一

個段落”Memory Manipulation and Control”，告訴我們 Step 2、Step 3 主要是在做記憶體位置的操作，ChatGPT 處理後的 Step 2 主要變成”the malware strategically modifies the program break location to 0x8fe7000”、而 Step 3 主要變成”the malware sets the thread local storage area to 0x8fe7860”，與原始的 Input 不會有太大的差異，因此基本上不會有錯誤發生，並且 ChatGPT 會依據正規英文文章的寫法將其增添連接詞、轉折語氣等英文補語，寫成一個完整的段落。

從 Fig.3.11 也能看出同樣的現象，圖中可以看到原始的 Input 中提到數次的”malware create child process: sh”，在 ChatGPT 理解後便會綜整成”the malware creates multiple child processes, each spawned using the ”sh” command”的寫法，同樣的因為原始的 Input 已經是具體的自然語言、且任務非常明確，就是去統整、分析各個步驟，因此 ChatGPT 的 Output 就不用做任何天馬行空的發想，而是產生出與原句相似 Output，並將其補全成為一個完整的段落，基本上也就沒有任何錯誤。因此透過這種具體、明確的 Input 及請求，ChatGPT 能清楚的了解自己的任務方向，不會有模糊指令時各種天馬行空的回應。

3.8 案例分析: Gafgyt

以此 ChatGPT Prompt 為基礎，本論文取 Gafgyt 為例進行實驗，結果如 Fig.3.15 3.16 3.17，從圖中可以觀察與 Dofloo 案例相同，1.ChatGPT 會將有關聯性的步驟綜整成一個一個段落，並以正規的英文形式撰寫，例如: Fig.3.16 中的 Step 20 - Step 28，惡意程式不斷的送出 singal，這一行為就被統整至”Signaling Processes and Waiting”這一段落。2.ChatGPT 會對專有技術名詞補充說明，例如 Fig.3.17 中惡意程式存取了各種與”watchdog”有關的檔案，ChatGPT 就會對此專有技術名詞補充說明，告訴我們這是一個與監控系統健康狀態有關聯的檔案，經

過人工的驗證也確定這是正確的資訊。

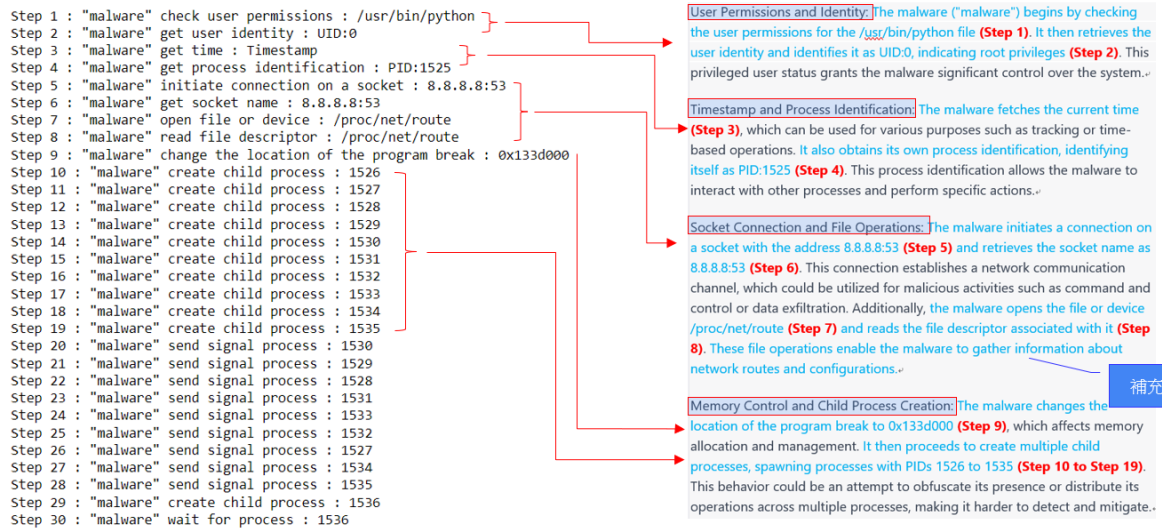


Figure 3.15: ChatGPT Gafgyt Example 1

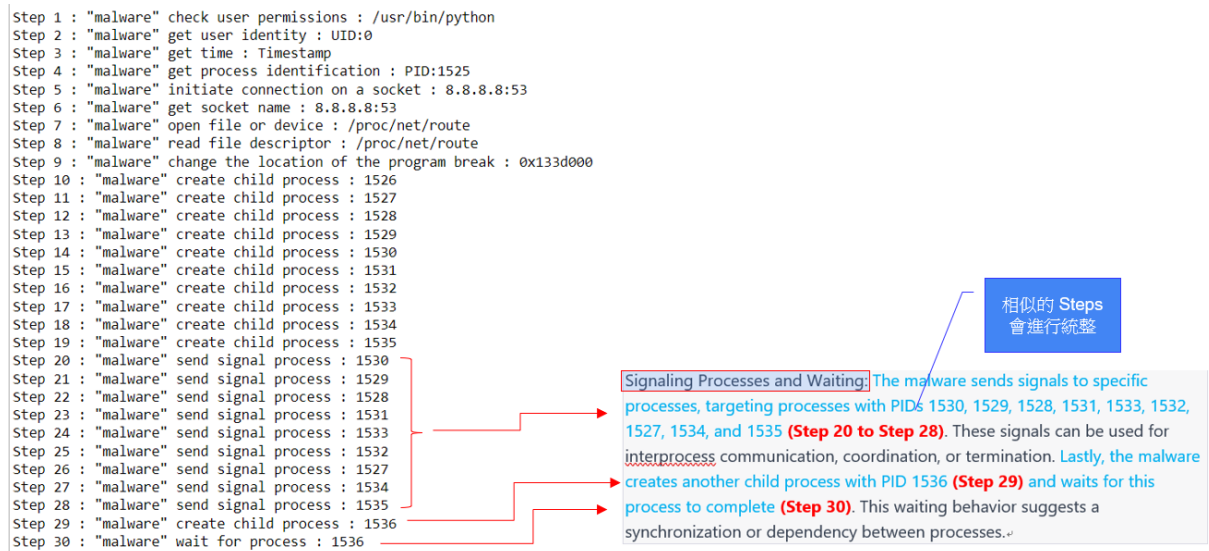


Figure 3.16: ChatGPT Gafgyt Example 2

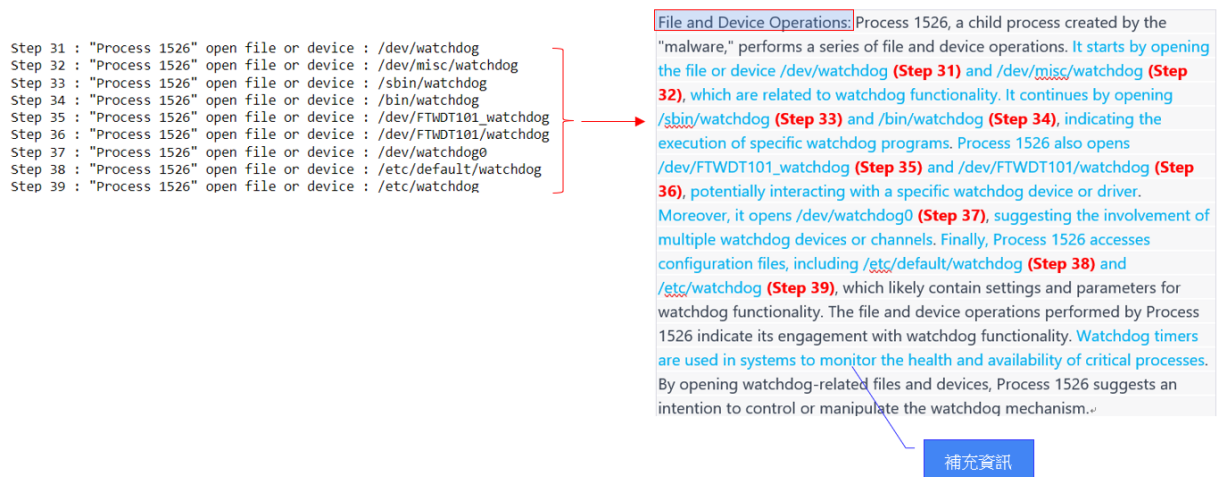


Figure 3.17: ChatGPT Gafgyt Example 3

3.9 案例分析: Darlloz

以相同的方式我們取 Darlloz 為例進行實驗，結果如 Fig.3.18 3.19 3.20 所示。從圖中可以觀察與前述案例相同，1.ChatGPT 會將有關聯性的步驟綜整成一個一個段落，並以正規的英文形式撰寫，例如: Fig.3.18 中的 Step 5 - Step 19 都是在對 process "sh" 進行操作，ChatGPT 就會綜整成一個段落"Process and Wait for "/bin/sh" 來表示。2.ChatGPT 會對專有技術名詞補充說明，例如 Fig.3.19 中的 Step 20 是對"/proc/ 這一資料夾做存取，ChatGPT 就會補充資訊告訴我們此資料夾是存有"information related to system processes."，也就是跟 process 有關的資訊。

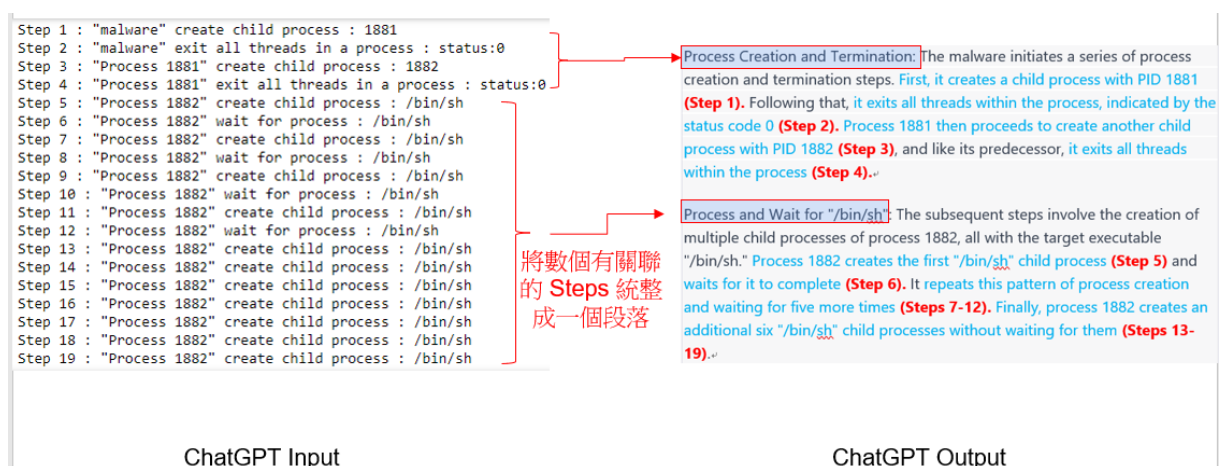


Figure 3.18: ChatGPT Dalloz Example 1

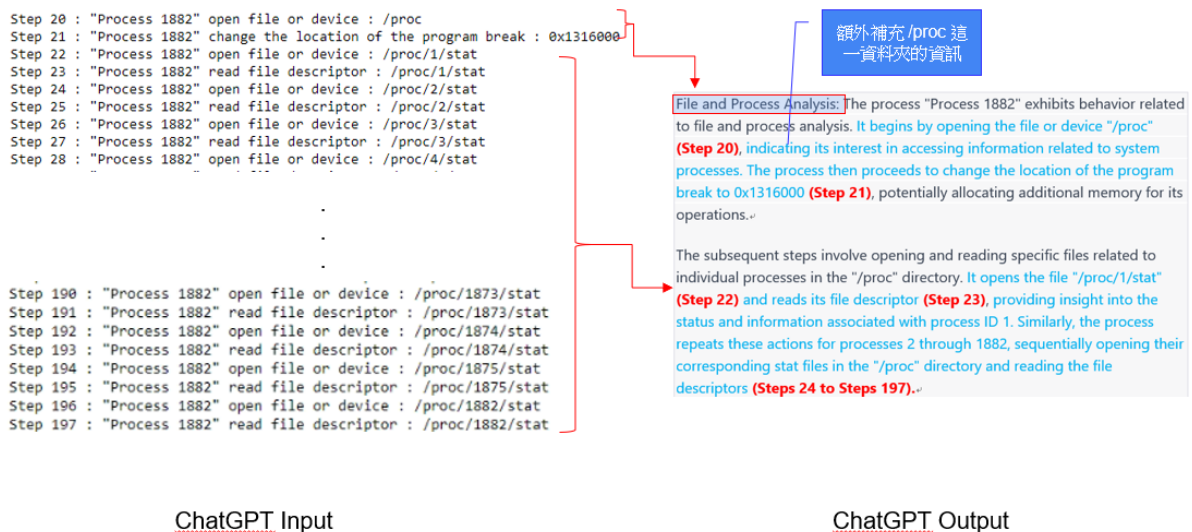


Figure 3.19: ChatGPT Dalloz Example 2

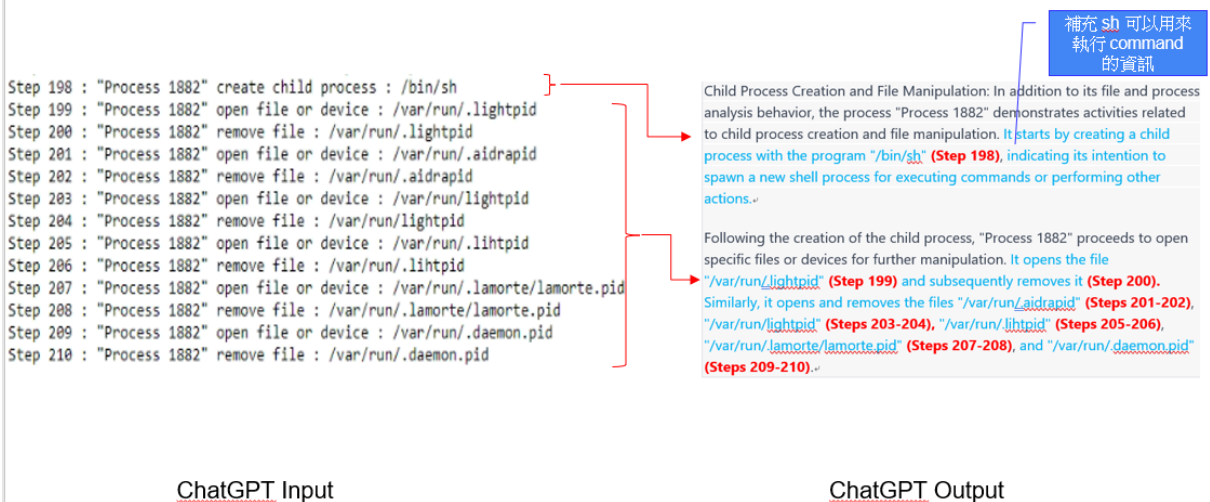


Figure 3.20: ChatGPT Dalloz Example 3

3.10 案例分析: Luabot

以相同的方式我們取 Luabot 為例進行實驗，結果如 Fig.3.21 3.22 3.23 所示。

從圖中可以觀察與前述案例相同，1.ChatGPT 會將有關聯性的步驟綜整成一個一個段落，並以正規的英文形式撰寫，例如:3.21 中可以看到 Step 1 - Step 3 都是在對記憶體位置、線程進行操作，ChatGPT 就會將其統整成一個段落”Memory and Thread Control”。2.ChatGPT 會對專有技術名詞補充說明，例如 Fig.3.22 中有提

到一個檔案”/dev/urandom”，我們直接看可能無法得它的作用，但 ChatGPT 就會補充資訊”commonly used to generate random data and can be leveraged by malware for various purposes, such as encryption or obfuscation”，告訴我們這個檔案是用來產生隨機資料，常被使用於加密、混淆等動作。

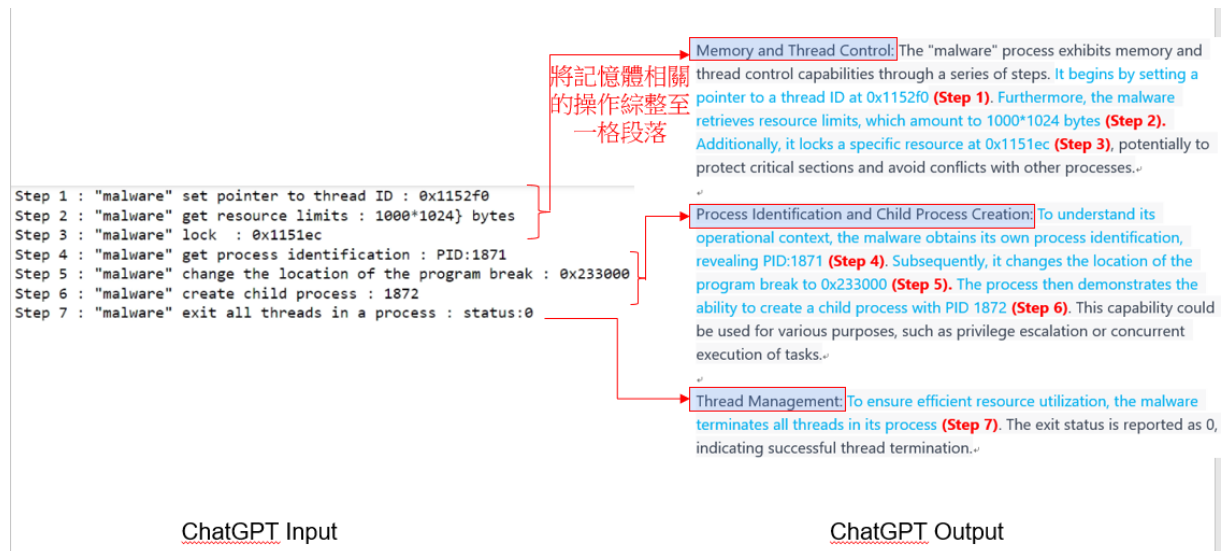


Figure 3.21: ChatGPT Luabot Example 1

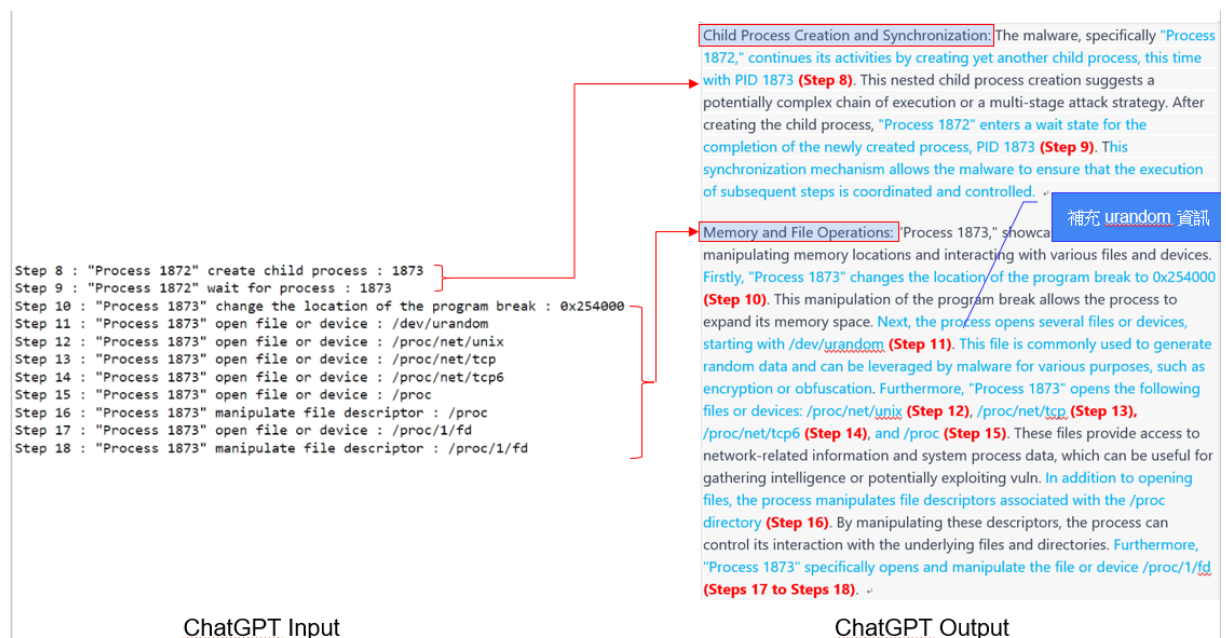


Figure 3.22: ChatGPT Luabot Example 2

```

Step 19 : "Process 1873" get file status : /proc/1/fd/0
Step 20 : "Process 1873" get file status : /proc/1/fd/1
Step 21 : "Process 1873" get file status : /proc/1/fd/2
Step 22 : "Process 1873" get file status : /proc/1/fd/3
Step 23 : "Process 1873" get file status : /proc/1/fd/4
Step 24 : "Process 1873" get file status : /proc/1/fd/5
Step 25 : "Process 1873" get file status : /proc/1/fd/6
Step 26 : "Process 1873" get file status : /proc/1/fd/7
Step 27 : "Process 1873" get file status : /proc/1/fd/8
Step 28 : "Process 1873" get file status : /proc/1/fd/9

```

```

Step 57 : "Process 1873" get file status : /proc/1/fd/46
Step 58 : "Process 1873" get file status : /proc/1/fd/47
Step 59 : "Process 1873" get file status : /proc/1/fd/48
Step 60 : "Process 1873" get file status : /proc/1/fd/49
Step 61 : "Process 1873" get file status : /proc/1/fd/57

```

相似的步驟
總整成一個段
落描述

File Status Checks Continuing its operations, "Process 1873" performs a series of file status checks on file descriptors associated with the /proc/1/fd directory. These checks provide insights into the status and availability of specific files within the system. The process begins by retrieving the file status of /proc/1/fd/0 through /proc/1/fd/57 (Step 19 ~ 61). The specific files associated with these file descriptors may vary and depend on the system's configuration and running processes. By inspecting the file status of these file descriptors, "Process 1873" gains insights into the availability and operational status of critical system resources. This information can be exploited to manipulate or interfere with the functioning of the associated files or to gather intelligence about the system's configuration.

ChatGPT Input

ChatGPT Output

Figure 3.23: ChatGPT Luabot Example 3

3.11 案例分析: Kaiten

以相同的方式我們取 Kaiten 為例進行實驗，結果如 Fig.3.24 3.25 3.26 所示。從圖中可以觀察與前述案例相同，1.ChatGPT 會將有關聯性的步驟綜整成一個一個段落，並以正規的英文形式撰寫，例如: Fig.3.26 中可以看到 Step 14 - Step 19 都在對各種檔案進行操作，ChatGPT 就會將其統整成一個段落"File Operations"；2.ChatGPT 會對專有技術名詞補充說明，例如 Fig.3.25 中的 Step 8 - Step 9 分別對"/etc/rc.d/rc.local (Step 8)"和"/etc/rc.conf (Step 9)"進行操作，ChatGPT 就會給我們補充資訊"These files are commonly associated with system initialization and configuration on Unix-based systems"告知我們這是一個 Unix 系統初始化時的配置檔案，更進一步的他還告訴我們"Modifying or inserting malicious code into these initialization scripts allows the malware to execute every time the system boots"，也就是可以透過修改這些配置檔案，來做到操控系統於開機時啟動自動執行惡意程式的動作。

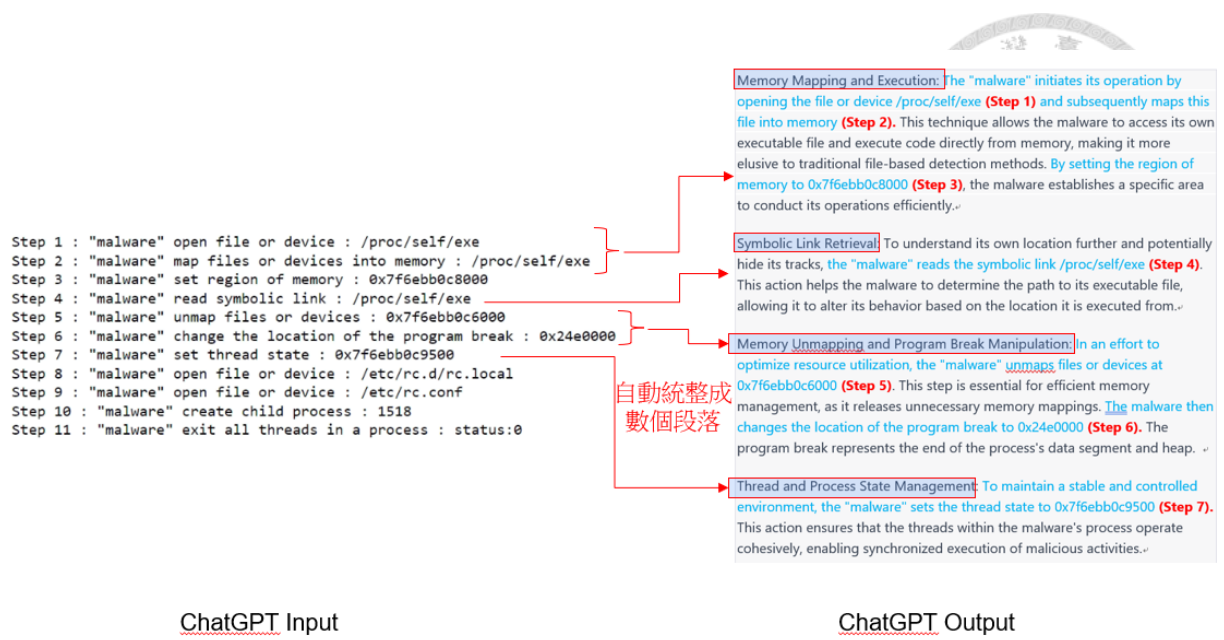


Figure 3.24: ChatGPT Kaiten Example 1

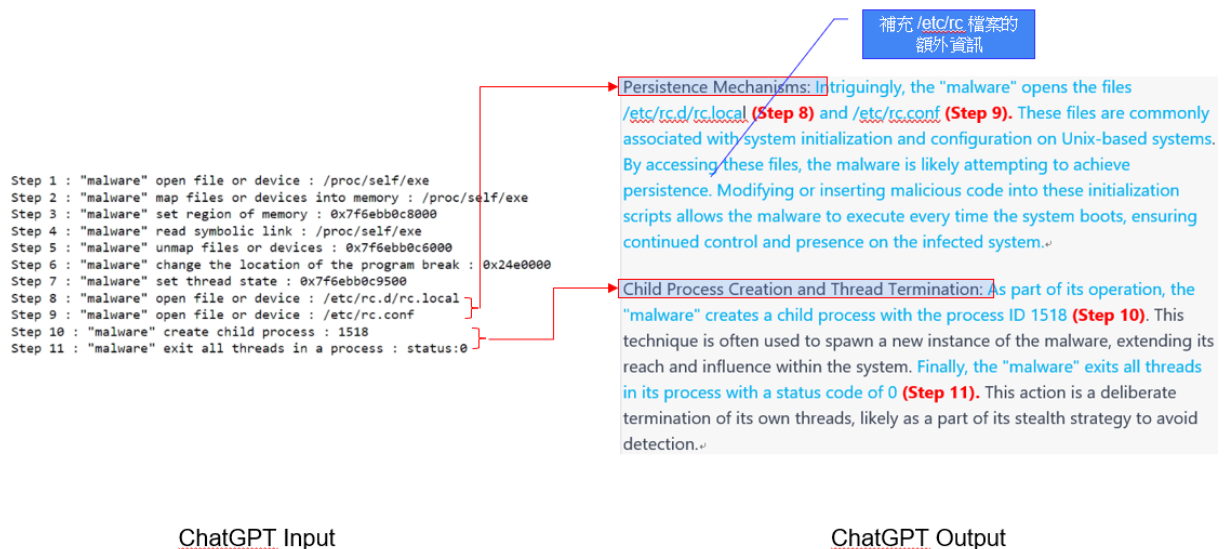


Figure 3.25: ChatGPT Kaiten Example 2

```

Step 12 : "Process 1518" get process identification : PID:1518
Step 13 : "Process 1518" get process identification : PID:1
Step 14 : "Process 1518" open file or device : /usr/dict/words
Step 15 : "Process 1518" get file status : /etc/resolv.conf
Step 16 : "Process 1518" open file or device : /etc/host.conf
Step 17 : "Process 1518" get file status : /etc/host.conf
Step 18 : "Process 1518" read file descriptor : /etc/host.conf
Step 19 : "Process 1518" open file or device : /etc/resolv.conf
Step 20 : "Process 1518" get file status : /etc/resolv.conf
Step 21 : "Process 1518" read file descriptor : /etc/resolv.conf
Step 22 : "Process 1518" execute program : uname

```

將相似的
步驟統整
成一個段落

Process Identification: The process "Process 1518" retrieves its own process identification, PID:1518 (Step 12), which helps in distinguishing it from other processes. Subsequently, it also obtains the process identification of the init process, PID:1 (Step 13). Understanding the process identification is vital for monitoring and controlling the behavior of the malware within the system.

File Operations: The process "Process 1518" performs various file-related operations in the system. It opens the file or device located at /usr/dict/words (Step 14). This action indicates potential interaction with the file's contents. Furthermore, the process retrieves the file status of /etc/resolv.conf (Step 15), which provides information about the file's attributes and permissions. "Process 1518" opens another file, /etc/host.conf. Like before, it proceeds to obtain the file status of /etc/host.conf and reads the file descriptor (Steps 16 to Steps 18), potentially extracting data from the file. Similar operations are repeated with /etc/resolv.conf (Steps 19 to Steps 21), indicating a focus on this specific file in the malware's activities.

Execution of External Programs: "Process 1518" executes the program uname (Step 22). This command provides information about the system and its kernel version.

ChatGPT Input

ChatGPT Output

Figure 3.26: ChatGPT Kaiten Example 3



Chapter 4 Conclusion

本論文的研究目標在於將一低階的 System Call Trace 繪製成惡意程式執行行為圖，並自動化產生高階自然語言惡意程式行為敘事文本，此舉對於想要瞭解惡意程式的人士，能夠提供一個非常實用的工具。

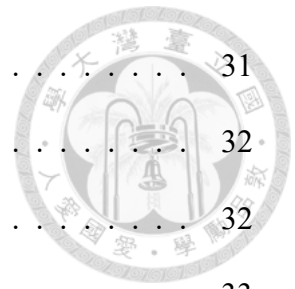
本論文的系統透過動態分析 (Dynamic Analysis) 的方式取得一惡意程式的 System Call Trace，我們針對每一個 Linux system call function 挑選出其動作 (operation) 下最重要的作用物件作為該動作動詞 (operator) 的受詞 (operand)。並以此為基礎繪製出惡意程式執行行為圖 (Attack Scenario Graph) 來完成視覺化的呈現。接著再透過建置 Natural Language Transformer 模組將 ASG 中以 System Call 組成的低階語言描述，經由 Natural Language Synonym Base 的輔助轉化為簡易的自然語言，此舉已能讓研究人員以相對高階的角度去分析惡意程式。因為此簡易的自然語言描述已經是精準的動作語義描述，所以在接下來透過 ChatGPT 輔助，將惡意程式各個步驟的簡易自然語言描述進一步綜整成一篇具高閱讀性與理解且包含背景與語義豐富的資訊文章，標示出重點並提供技術用語的補充資訊，用以提供研究人員更直觀且豐富的資訊。



List of Figures

2.1	ASG Structure Example	6
3.1	System Arch Automatic Narrative Essay Generation	13
3.2	Memory Reduction	16
3.3	Time Reduction	16
3.4	NIC Reduction	17
3.5	Steps Reduction	17
3.6	ASG Step Example	19
3.7	ASG Steps List Example	19
3.8	Linux man page example	20
3.9	Natural Language (NL) Transformer Example	23
3.10	ChatGPT Dofloo Example 1	24
3.11	ChatGPT Dofloo Example 2	25
3.12	ChatGPT Dofloo Example 3	25
3.13	ChatGPT Prompt 1	27
3.14	ChatGPT Prompt 2	27
3.15	ChatGPT Gafgyt Example 1	30
3.16	ChatGPT Gafgyt Example 2	30
3.17	ChatGPT Gafgyt Example 3	31

3.18 ChatGPT Dalloz Example 1	31
3.19 ChatGPT Dalloz Example 2	32
3.20 ChatGPT Dalloz Example 3	32
3.21 ChatGPT Luabot Example 1	33
3.22 ChatGPT Luabot Example 2	33
3.23 ChatGPT Luabot Example 3	34
3.24 ChatGPT Kaiten Example 1	35
3.25 ChatGPT Kaiten Example 2	35
3.26 ChatGPT Kaiten Example 3	36





List of Tables

3.1	Dofloo: Scale Reduction	16
3.2	Xorddos: Scale Reduction	17
3.3	Darlloz: Scale Reduction	17
3.4	LuaBot: Scale Reduction	18
3.5	Kaiten: Scale Reduction	18
3.6	NL Synonym base schema	21



References

- [1] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In 2020 IEEE Symposium on Security and Privacy (SP), pages 1172–1189, 2020. doi: 10.1109/SP40000.2020.00096.
- [2] Jambit GmbH. Linux man page: clone(), 2023-03-30. URL <https://man7.org/linux/man-pages/man2/clone.2.html>.
- [3] Jambit GmbH. Linux man page: system call, 2023-06-24. URL <https://man7.org/linux/man-pages/man2/syscalls.2.html>.
- [4] Zhenyuan Li, Jun Zeng, Yan Chen, and Zhenkai Liang. Attackg: Constructing technique knowledge graph from cyber threat intelligence reports. In Lecture Notes in Computer Science, page 589–609, 2022. doi: 10.1007/978-3-031-17140-6_29.
- [5] Sadegh M. Milajerdi, Rigel Gjomemo, Birhanu Eshete, R. Sekar, and V.N. Venkatakrishnan. Holmes: Real-time apt detection through correlation of suspicious information flows. In 2019 IEEE Symposium on Security and Privacy (SP), pages 1137–1152, 2019. doi: 10.1109/SP.2019.00026.
- [6] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—a state of the art survey. In ACM Comput. Surv., Vol. 52, No. 5, Article 88, 2019. doi: 10.1145/3329786.

- [7] ZHI-KANG, CHEN. Chatgpt case study, 2023-08-04. URL https://drive.google.com/drive/folders/1eEZe0ZwsiNfXoIsxm45scLTq5faCat-D?usp=drive_link.

