國立交通大學

資訊科學與工程研究所

碩士論文

符號環境建置與調適性符號輸入選擇

Symbolic Environment Construction and Adaptive Symbolic Input Selection

研究生：呂翰霖

指導教授：黃世昆　教授

中華民國一百零一年七月

符號環境建置與調適性符號輸入選擇

學生：呂翰霖　　指導教授：黃世昆

國立交通大學資訊科學與工程研究所碩士

摘要

由於軟體品質的良莠不齊，使得軟體漏洞不斷被發現。而在現今的社會中，這些漏洞甚至可以危害到公共基礎建設進而影響人身安全，而一個有效率的軟體檢測方法可以在這些漏洞發生問題之前預先提防。

因此在本篇論文我們提出一系列的方法來加速軟體的檢測並藉由產生漏洞的脅迫(exploit)來提醒軟體開發者能夠優先修補較危險的漏洞以減少危險的發生。我們提出的方法分別是：符號汙染分析、符號環境建置以及選擇性輸入，並運用這些方法實作改良在原有之軟體品質與自動脅迫產生平台：CRAX 上，第一個方法是基於符號執行(symbolic execution)的汙染分析(taint analysis)，主要用來快速的釐清 crash 檔案對漏洞程式的影響，而第二個方法將建置一個符號環境，讓 CRAX 能夠做到 binary level(執行檔)的測試，選擇性輸入讓 CRAX 在自動產生脅迫上以及測試大型程式更有效率。我們將改良過後的 CRAX 與現有同質性之工具作比較，皆有優良的成績，證明了這些方法在實際應用上的可行性。

誌謝

首先感謝我的指導教授黃世昆老師，帶領我一起研究軟體安全這個神秘的領域讓我一步一步的學會了很多知識，除了研究上的指導，並時常關心我的生活與課業，時時讓我感到溫暖。

感謝我的父母，你們給我一個美好的家庭，讓我在充滿愛的環境裡長大，今天走到這一步全是因為你們努力的栽培，你們是我的榜樣，我永遠愛妳們。感謝若望，為了做你的表率讓我更成熟了，雖然我們差了六歲，但還是能玩在一塊，每次回家看到你我就開心，還好有個弟弟。

感謝實驗室的學長們佑鈞、世欣作為我的榜樣，引領我進入實驗室生活，還有博彥還有孟緯，有你們在的兩年生活真的非常有趣，感謝銘祥學長在研究上給我很多的幫助，以及肇鈞學長對此篇論文的建議使其更完整，還有俊維，韋翔、基傑、偉明、奕任，大家一起奮鬥，一起歡樂的日子是我最珍貴的回憶，也感謝實驗室的學弟妹們，實驗室因為你們而更有朝氣。

感謝家族裡的長輩們與堂表兄弟姊妹，感謝你們的鼓勵與祈禱，讓我時時充滿力量，感謝成長的路上所有幫助過我的人，是你們的幫助才有今天的我。

感謝天主，祢時常磨練我的心智，使我茁壯，儘管過程辛苦，但祢每一次都陪我度過，感謝讚美祢。

內容

簡介 1

1.1 背景 1

1.1.1 汙點分析(Taint Analysis) 1

1.1.2 符號執行(Symbolic Execution) 2

1.1.3 擬真執行(Concolic Execution) 4

1.1.4 單一路徑擬真執行(Single Path Concolic Execution) 5

1.1.5 符號環境 6

1.1.6 自動脅迫產生(Automatic Exploit Generation) 7

1.1.7 Crash 類型 7

1.2 研究動機與目標 8

1.2.1 符號環境建立很困難 8

1.2.2 CRAX 在大型程式上的分析沒效率 8

相關研究 11

2.1 符號執行 11

2.2 可脅迫性偵測(Exploitable Detect) 12

2.3 脅迫產生(Exploit Generation) 12

2.4 綜合比較 13

研究方法 14

3.1 程式碼的符號執行 14

3.2 代理程式與共享符號記憶體 15

3.3 Symbolic File 建置 15

3.4 Symbolic Socket 建置 16

3.5 Symbolic Environment Variables 建置 17

3-6 Symbolic Standard Input 建置 17

3.7 Symbolic Argument Variables 建置 18

3.8 調適性符號輸入選擇(Adaptive Symbolic Input Selection) 18

3.8.1 符號汙染分析(symbolic taint analysis) 18
3.8.2 選擇性輸入(input selection) 20
3.8.3 調適性符號輸入選擇的特殊情況 22
實作 24
4.1 S2E API 介紹 24
4.2 Symbolic File 25
4.3 Symbolic socket 26
4.3.1 Symbolic Server 26
4.3.2 Symbolic Client 26
4.4 Symbolic Environment Variables 27
4.5 Symbolic Standard Input 28
4.6 Symbolic Argument Variables 29
4.7 S2E 的記憶體架構 30
4.8 符號汙染分析 31
結果與實驗 32
5.1 實驗環境 32
5.2 新舊 CRAX 效能比較 32
5.3 與 AEG 標竿程式的比較 34
5.4 與 MAYHEM 的比較 35
5.5 測試大型程式 36
5.6 小結 36
結論與未來研究 37
6.1 結論 37
6.2 未來研究 38
6.2.1 符號環境的改進 38
6.2.2 調適性符號輸入選擇的改進 38
參考文獻 39
附錄 A 長條圖數據 41

程式碼目錄

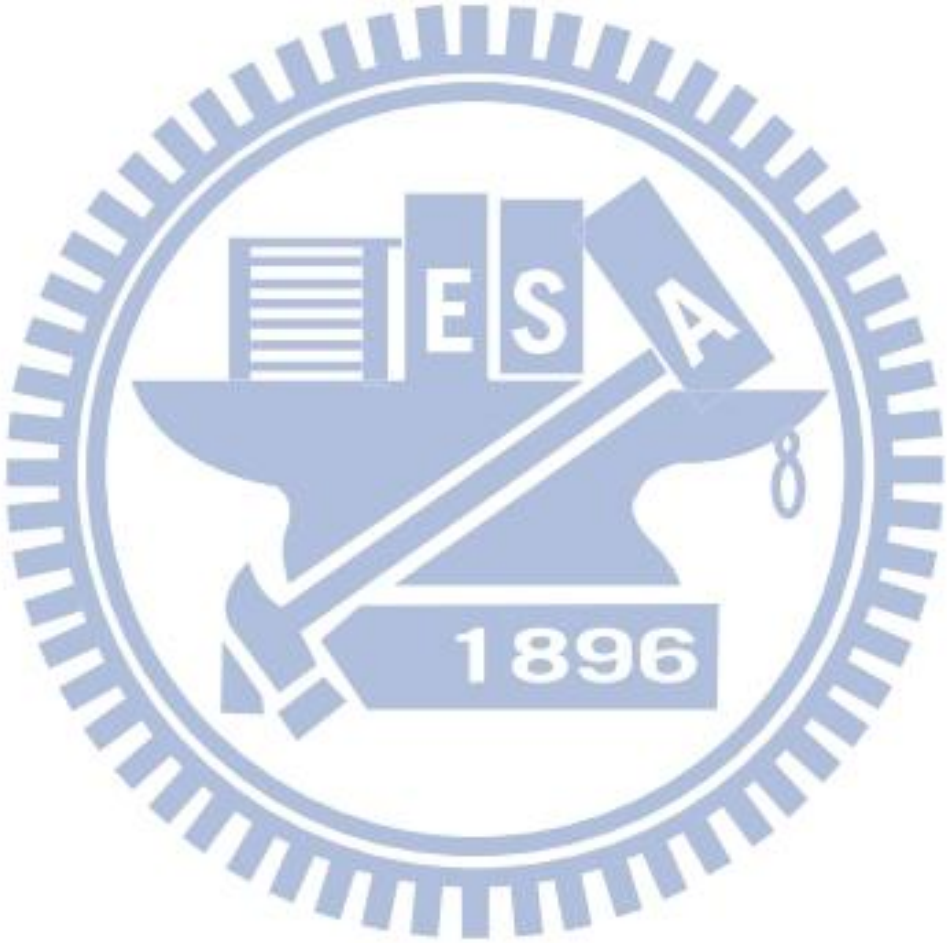
程式碼 1 符號執行範例 3
程式碼 2 擬真執行範例 5
程式碼 3 符號執行比較組 14
程式碼 4 符號執行對照組 14
程式碼 5 TYPE 1 CRASH 23
程式碼 6 SYMBOLIC FILE 25
程式碼 7 SYMBOLIC SERVER 26
程式碼 8 SYMBOLIC CLIENT 27
程式碼 9 SYMBOLIC ENVIRONMENT VARIABLES 28
程式碼 10 SYMBOLIC STANDARD INPUT 29
程式碼 11 SYMBOLIC ARGUMENT VARIABLES 30

圖表目錄

圖表 1 符號執行流程圖 4
圖表 2 擬真執行流程圖 5
圖表 3 單一路徑擬真執行流程圖 6
圖表 4 AUTOMATIC EXPLOIT GENERATION 流程圖 7
圖表 5 符號輸入長度與路徑探索 9
圖表 6 符號輸入長度與脅迫產生 9
圖表 7 共享符號記憶體 15
圖表 8 檔案對應至記憶體中 16
圖表 9 SYMBOLIC CLIENT 17
圖表 10 SYMBOLIC SERVER 17
圖表 11 SYMBOLIC STANDARD INPUT 18
圖表 12 當 EIP 暫存器被符號變數汙染，可以直接知道來源與目標對應的關係 19
圖表 13 選擇性輸入流程圖 21
圖表 14 堆疊溢位 23
圖表 15 物件 OBJECTSTATE 資料結構 31
圖表 16 新舊 CRAX 長條圖數據 41
圖表 17 使用長條圖與 AEG 41
圖表 18 使用長條圖與 MAYGEM 比較 42

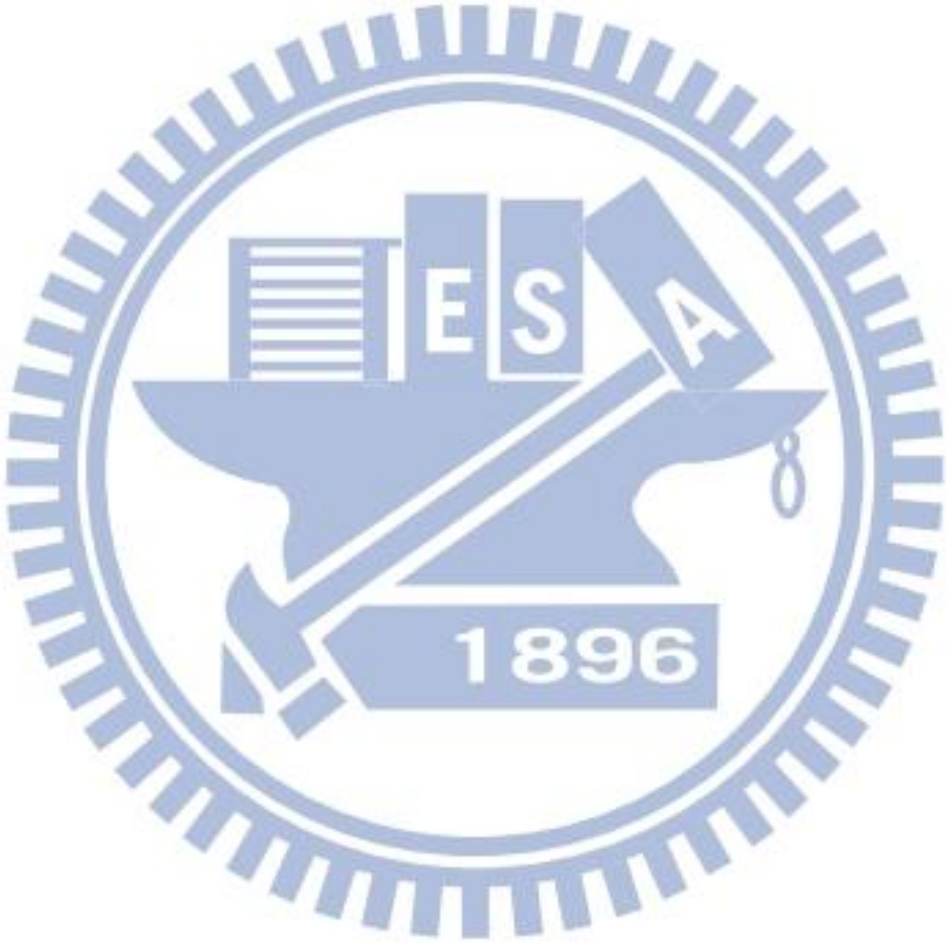
表格目錄

表格 1 符號長度與 CRAX 執行時間 9
表格 2 綜合比較 13
表格 3 新舊 CRAX 比較 33
表格 4 AEG 標竿程式 35
表格 5 與 MAYHEM 比較 36
表格 6 大型程式測試數據 36



演算法目錄

演算法 1 符號汙染分析的演算法.....20
演算法 2 選擇性輸入.....22



第一章

簡介

近年來儘管軟體安全漸漸受到重視，軟體被駭客攻擊的事件仍層出不窮，關鍵在於軟體安全漏洞很多，但可以產生脅迫(exploit)的確很少，攻擊者只要發現一個 0-day 就可以為所欲為，而反觀守方的資安專家們，必須分析所有的軟體漏洞才有可能杜絕威脅，然而隨著現在軟體工業的發展，軟體規模不斷擴大，設計也越來越複雜，靠人力分析一個軟體漏洞少則花費數天，多則數禮拜甚至數個月，分析專家們，窮盡所有心力也難以阻擋不法者的攻擊，而大型公司與一般使用者更束手無策。也因如此一個新興的技術逐漸受到重視，自動脅迫產生(automatic exploit generation)：藉由自動產生軟體漏洞的脅迫(exploit)，來證明漏洞是否可被攻擊。此技術仍然在開發中，對於大程式的支援，以及測試 binary 檔(執行檔測試)的效率，都有其侷限性，若能在這兩方面有所突破，或許軟體安全攻防的平衡，將有所改變。

1.1 背景

1.1.1 汙點分析(Taint Analysis)

汙點分析[1]是一種很常見的動態分析技術，通常被應用在找尋軟體的安全

漏洞、分析惡意軟體以及測資產生等領域上。主要的分析方法，是藉由檢查系統中記憶體或暫存器等是否有被資料流所影響，被影響的地方就是被汙染了(tainted)。但是這樣的分析有著一個很大的缺陷，就是即使知道問題點有被資料流影響，但是並不能知道被汙染處與資料流之間的對應關係。

1.1.2 符號執行(Symbolic Execution)

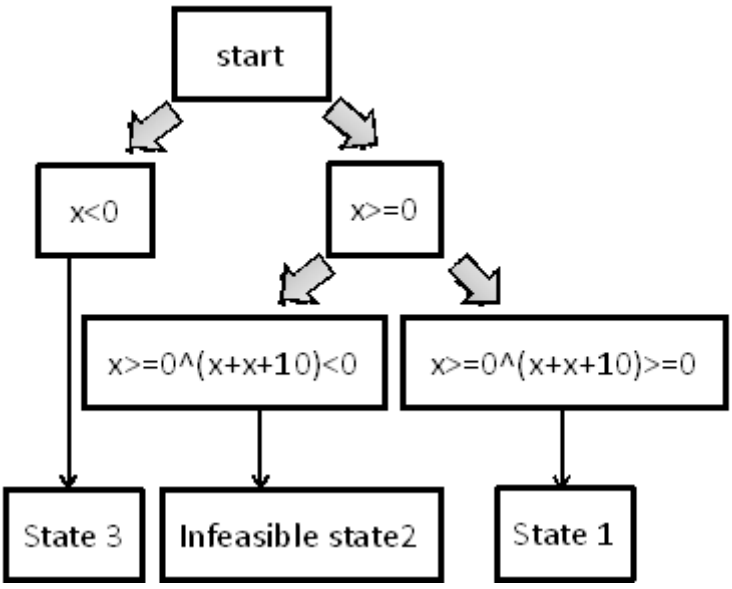
符號執行[2,3]也是另外一種常見的動態分析用來測試程式的路徑覆蓋率(path coverage)，符號執行將受測程式的資料流用符號變數(symbolic variable)來代替，符號變數並沒有特定的值，可以想像成是一個值域。當普通變數存取符號變數時，也會變成一個符號變數。

在符號執行中，當符號變數進入到一個分支條件(branch condition)，會有兩個新的狀態產生並分別對應兩組相反的限制式(constraint)，一個組滿足分支條件(branch condition)，另一組則無法滿足，而這兩組限制式則被稱為路徑限制式(path constraint)分別代表不同的程式執行路徑。當符號變數再次進入分支條件(branch condition)時，產生的新限制式如果滿足原本的路徑限制式則會合併成新的路徑限制式，如果不能滿足，則被丟棄，代表這一條路徑已經走到頂端。符號執行完畢後，路徑限制式代表唯一的一條程式執行路徑，並且可以透過 solver 產生出一組可以來到相同路徑的測資(test case) 考慮以下程式碼

```
1.  func(int x ){
2.      if(x>=0){
3.          y = x + 10
4.          if(x+y>=0){
5.              printf("state 1");
6.          }else{
7.              printf("state 2")
8.          }
9.      }else{
10.         printf("state 3");
11.   }
12. }
```

程式碼 1 符號執行範例

在第一行 x 為符號變數，當程式執行到第二行時，碰到程式分支點(branch)，這時候會程式分兩種不同的狀態繼續執行，一個代表 x>=0 ，而另一個為 x<0 ，在第三行，一個非符號變數 y 被 x 存取，所以 y 也變成符號變數，它代表 x + 10。當程式執行到第五行時又碰到了另一個分支點，程式分為兩個相反的路徑，一個代表 x>=0^(x+x+10)<0 這條路徑，另一條路徑以 x>=0^(x+x+10)>=0 來表示，但由於 x>=0^(x+x+10)<0 是一條矛盾的限制式因此會被符號執行捨去。程式執行完畢後，擬真執行總共衍生了兩條限制式，分別代表不同的執行路徑，而這兩條分別可以到達 state1 和 state3，圖表一為完整的執行流程。



圖表 1 符號執行流程圖

1.1.3 擬真執行(Concolic Execution)

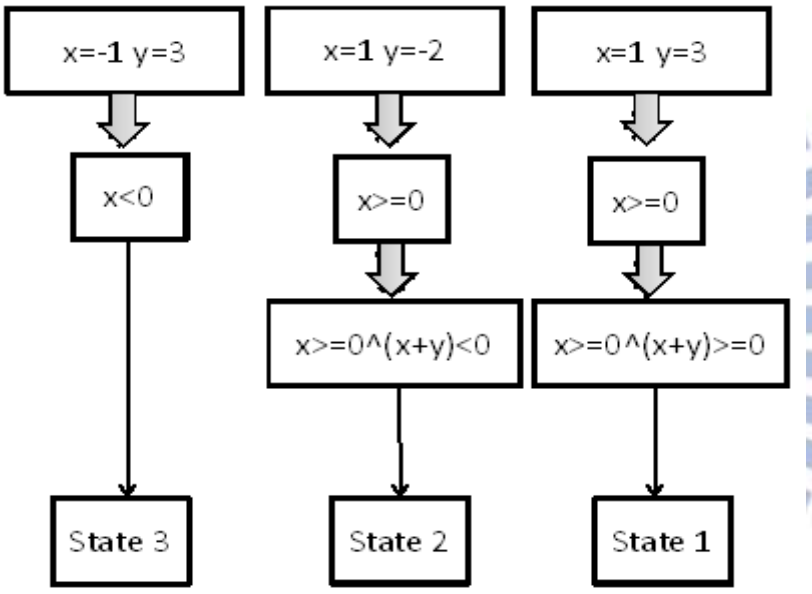
符號執行會有無限迴圈的問題，例如 GUI 程式需要無限迴圈來等待新事件的發生，這時候符號執行就會產生無限的路徑，因此，符號執行就必須給定一個迴圈的上限，或一個路徑選擇的策略，而擬真執行就是一種路徑選擇的方法。

擬真執行[4]包含具體執行(concrete execution)以及符號執行(symbolic execution)的特性，因此可避免符號執行有無限迴圈的問題，也比具體執行有著更高的路徑覆蓋率(path coverage)。擬真執行以一組隨機的輸入開始執行(input)，並在執行過程中記錄路徑限制式(path constraint)，當執行完畢時，根據否定(negiate)路徑限制式裡的分支限制式(branch constraint)產生新的測資，直到全部路徑走完為止。以下以實際的例子來說明。考慮 程式碼 2

程式一開始 x 與 y 為符號變數，並且初始值 x=-1 y=3，當執行到第二行時，碰第一個分支點，符號變數 x=-1 不滿足限制式 x>=0，因此 x<0 被記錄到路徑 1，程式繼續執行到第八行，來到了 state 3，結束了第一次執行，擬真執行根據否定(negate)路徑 1 的限制式也就是 x>=0 來產生新的一組輸入(input) 這邊的例子是 x=1，y=3，然後重新執行，當成是執行到第 2 行限制式被加入到 path2，當程式執行到第三行時，x=1、y=3 滿足限制式 x+y>=0 於是限制式 x+y>=0 也被加入倒 path2，接著結束這次的執行，擬真執行又根據否定 path2 得到第三組輸入，此處的例子是 x=1、y=-2，重新執行後執行過程中蒐集到的限制式為 x>=0^(x+y)<0，代表 path3，這時候所有路徑皆已到達，結束擬真執行。圖表二為執行過程的流程圖。

```
1.  func(int x  int y){
2.      if(x>=0){
3.          if(x+y>=0){
4.              printf("state 1");
5.          }else{
6.              printf("state 2")
7.          }
8.      }else{
9.          printf("state 3");
10.   }
11. }
```

程式碼 2 擬真執行範例

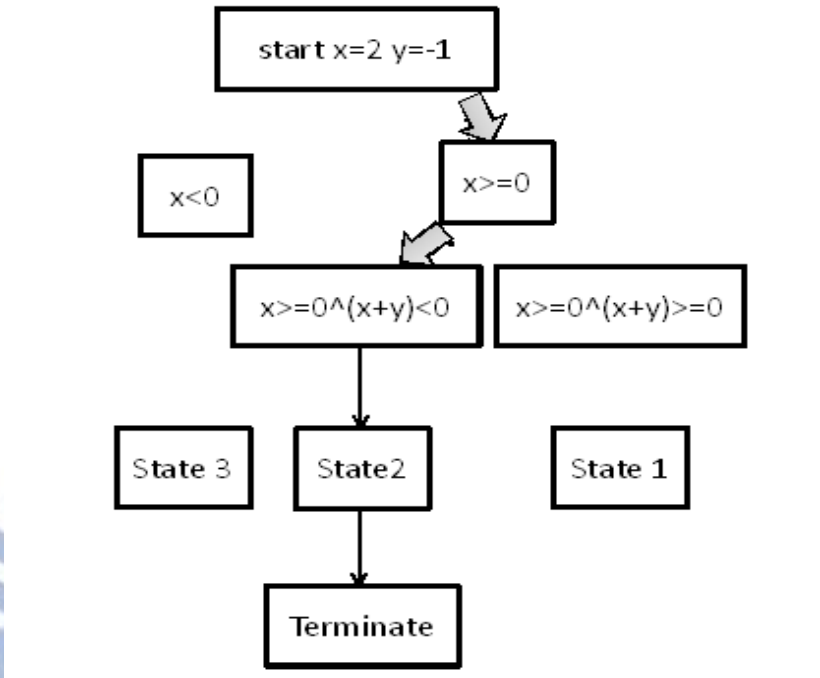


圖表 2 擬真執行流程圖

1.1.4 單一路徑擬真執行(Single Path Concolic Execution)

不同於符號執行與擬真執行，單一路徑擬真執行的目的不是在於覆蓋率的檢測而針對於蒐集目的路徑途中經過的控制流程(control-flow)資訊。以程式碼 2 為例，單一路徑擬真程式一開始時 x=2，y=-1，當執行到第一個分支點後，路徑限制式為 x>=0，接著遇到第二個分支點，x 與 y 滿足 x+y >0，因此路徑限制式變為 x>=0^(x+y)<0 程式執行到 state2 後單一擬真執行結束測試不再搜索其他程

式執行路徑。圖表三為單一路徑擬真執行的流程。



圖表 3 單一路徑擬真執行流程圖

1.1.5 符號環境

符號環境最早是由 KLEE[5]所提出，目的是在做符號執行(symbolic execution)測試時，盡可能的視所有電腦上的資料流為符號輸入(symbolic input)來達到測試沒有原始碼的執行檔。常見的符號環境有五種：

Symbolic file：將檔案視為為符號輸入，也就是受測程式可以透過讀取檔案的方式來執行符號測試。

Symbolic socket：將所有的網路資料流視為符號輸入

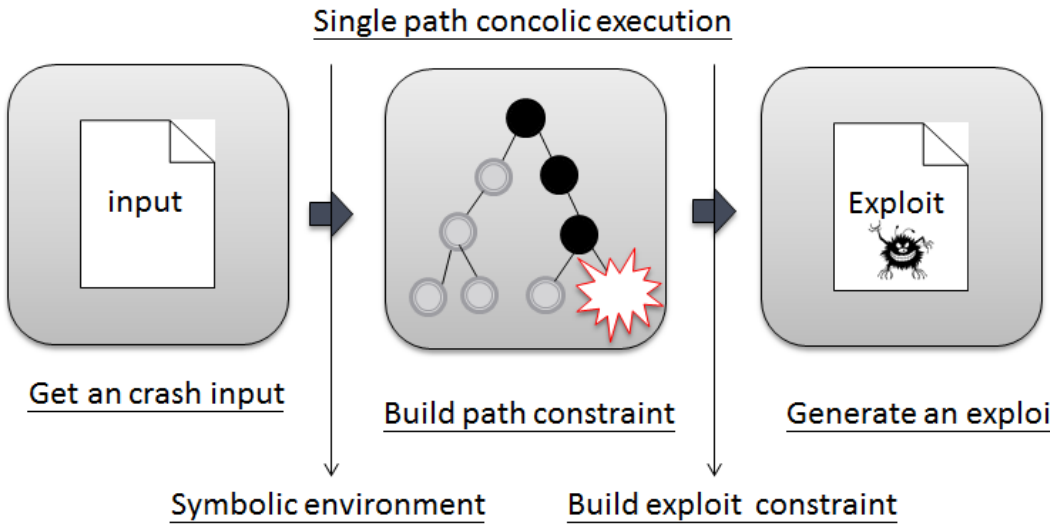
Symbolic standard input：將標準輸入視為符號輸入

Symbolic environment variable：將系統環境變數視為符號輸入

Symbolic argument variable：將執行程式參數視為符號輸入

1.1.6 自動脅迫產生(Automatic Exploit Generation)

CRAX[13]裡面提到自動脅迫產生分成三個階段，第一：取得一個受測程式的 crash 檔案，第二：透過符號環境(symbolic environment)來執行單一路徑擬真執行(single path concolic execution)，第三：建立脅迫限制式(exploit constraint)並透過 solver 產生脅迫(exploit)。圖 4 為完整流程。



圖表 4 Automatic Exploit Generation 流程圖

在第一步驟，可以利用 fuzzer 工具來產生 crash 檔案，或是藉由人為測試來產生，而第二步驟主要是收集執行時的單一執行路徑限制式，這條路徑可以保證觸發受測程式發生異常。第三步驟，脅迫限制式包刮在第二部分裡收集到的路徑限制式，以及 shellcode [1] 和 shellcode 所在的位置，還有 NOP sled。最後 solver 再根據這些限制式來產生脅迫。

1.1.7 Crash 類型

當程式執行的過程中，如果不合法的記憶體位置被讀、或存取、執行，就會發生異常，而針對執行異常本篇論文將其分為兩類：

[1] 一般泛指一段可以開啟 command shell 的程式碼

類別一：EIP[2]暫存器的值直接對應到資料流的某一區塊，例如堆疊緩衝區溢位，通常只是將資料流的一個區段直經過記憶體後被存到 EIP 暫存器。

類別二：EIP 暫存器的直間接對應到資料流的某一區塊例如整數溢位 Integer overflow，heap overflow，uninitialized variables

1.2 研究動機與目標

1.2.1 符號環境建立很困難

KLEE 的符號環境建置使用 2500 行程式碼來修改 40 個函式呼叫(function call)，而 AEG [6] 則花了 5000 程式碼來修改 70 個函式呼叫(function call)，MAYHEM [7] 則重新改寫 42 個函式呼叫總共寫了了上萬行的程式, Hellen[8]為了達到執行檔的污染分析(taint analysis)用了 7000 行程式碼來攔截以及改寫系統呼叫(system call)。上面提到的做法都運用修改(函式呼叫)function call 或是攔截系統呼叫(system call) 的方法來達到符號環境建置，但是這兩種方法不只費時也費人力，所以在這篇論文當中，想要嘗試一種簡單的方法來達到符號環境的建置使得 binary level 的符號執行更簡單。

1.2.2 CRAX 在大型程式上的分析沒效率

CRAX 在執行大型程式的時候顯得沒有效率，原因可分成兩類如下

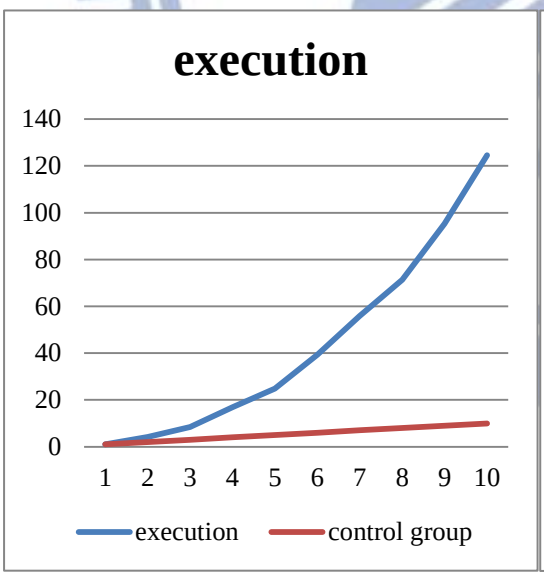
全環境(whole-system)的符號執行：由於全境環境的符號執行，必須在執行過程中記錄額外的資料，以及花更多地執行時間在核心(kernel)，這使得 CRAX 在執行大程式的時候，越為不利。

[2] 指向待執行的 instruction

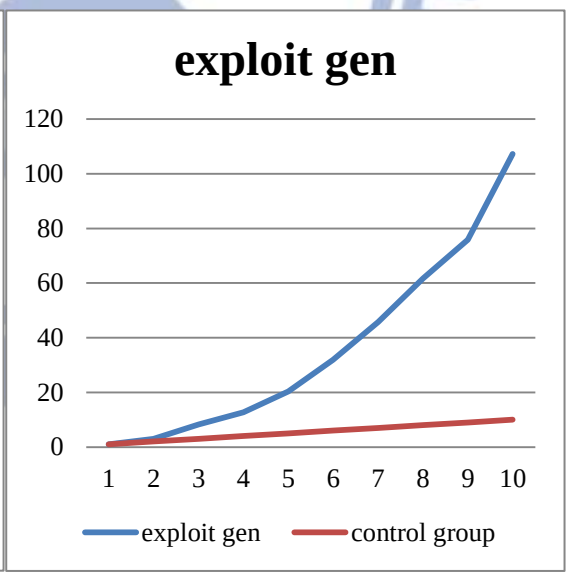
符號輸入長度(symbolic input length)影響執行時間：圖表 4 與圖表 5 顯示出當符號長度呈線性成長，但是單路徑擬真執行時間以及脅迫(exploit)產生時間卻為指數成長。

Symbolic input length(byte)	Explore time (s)	Exploit gen time (s)
300	3.76	6.96
600	15.45	20.69
900	31.9	57.43
1200	63.9	88.4
1500	94.3	141.6
1800	148.67	222.02
2100	211.5	318.29
2400	270.1	430.3
2700	360.56	528.65
3000	470.75	746.27

表格 1 符號長度與 CRAX 執行時間



圖表 5 符號輸入長度與路徑探索



圖表 6 符號輸入長度與脅迫產生

儘管全環境的符號執行降低了 CRAX 的執行效率，但是為了能夠針對跨平台程式的檢測，這是必須接受的妥協。

假設一個自動脅迫產生的情境，crash input 的大小為 5000byte ，那麼 CRAX 將會花費難以接受的時間在執行單一路徑 擬真執行，以及脅迫產生上，但是事實上一個脅迫的大小，在真實環境的限制下，通常都很小，事實上，產生一個 exploit 首要關心的是，input 如何影響 EIP，因此若是能夠很快速的發現 input 中，直接影響 EIP 的位元組(critical byte)，則就可以根據 critical byte 來決定符號輸入的常常度以及位置，來達到增加 CRAX 在執行上的效率。

本篇論文將提出一個方法能夠快速地找尋 critical byte ，並且提出一個策略利用 critical byte 來決定符號輸入的位置與長度。

第二章

相關研究

本節分四部分，第一部分列出一些運用擬真或是符號執行來做漏洞偵測，或是路徑覆蓋率偵測的工具，第二部分介紹的工具主要是用在判斷一個可以造成軟體崩壞(crash)檔案是否是可脅迫的(exploitable)，第三部分介紹現有的自動脅迫產生(automatic exploit generation)工具。第四部份以一個表格來呈現現有工具對符號環境支援度做相關的比較。

2.1 符號執行

Catchconv [9]可以偵測軟體中是否有整數溢位的問題，並且嘗試產生一個可觸發此漏洞的檔案。主要的方法是先用擬真執行來到達一個特定的目的(可能發生整數溢位的地方)在透過執行過程中所收集的路徑限制式(path constraint)去計算出一個可以觸發整數溢位的輸入。

KLEE是一個基於LLVM[12]的軟體測試平台，主要運用符號值執行來進行程式路徑覆蓋(path coverage)測試以及測資產生(test case generation)。另外KLEE也是最早提出符號環境建置的工具.

S^2E [10] 是一個分析軟體行為的平台，並建置在 QEMU[13]以及 KLEE 上支

援(whole-system)全系統的符號執行和擬真執行而且提供完善的符號環境。

2.2 可脅迫性偵測(Exploitable Detect)

Bitblaze[11]是一個執行檔分析的平台，並且支援許多常見的軟體分析技術，例如fuzzing，動態汙點分析(dynamic taint analysis)，符號執行，以及靜態分析。Bitblaze也被應用在crash 檔案的檢測，它可以判斷crash檔案是否可脅迫(exploitable)，並且找出系統發生crash 的真正原因。

!Exploitable[12] 是一個微軟除錯器(debugger)的套件，可以動態的檢測軟體漏洞是可被攻擊，並將受測程式分級為”exploitable”, “probably exploitable”, “probably not exploitable”, 或 “unknown”。

2.3 脅迫產生(Exploit Generation)

Heelan 是第一個自動產生脅迫的工具，主要的方法是藉由汙染分析來偵測被汙染的EIP暫存器，並透過執行過程所收集的控制流程(control-flow)資訊來產生脅迫。另外Heelan可以直接對執行檔進行檢測，但是他必須攔截系統呼叫(system call)才能做到汙染分析。

AEG (automatic exploit generation) 是第一個點對點(end-to-end)自動產生脅迫的以及漏洞探測的工具，主要結合靜態分析、擬真、符號執行技術，雖然AEG有提供符號環境，但是仍然需要受測程式的原始碼才能夠產生脅迫。

MAYHEM是第一個點對點(end-to-end)且不需要受測程式原始碼就可以自動產生脅迫的工具，並且運用混和擬真以及符號執行的技術來增加自身效率。除此之外MAYHEM支援五項符號環境，以及Linux 和Windows的軟體檢測。

CRAX[13] 是交通大學軟體品質實驗室發展的工具，對於一個可脅迫的crash檔案，可以自動產生出受測程式的脅迫，而不需要受測程式的程式碼，並且同時

支援Linux 以及Windows平台的測試。CRAX能夠完全支援常見的符號環境，以及在大程式的測試也有不錯的成績。

2.4 綜合比較

System	Exploit-gen	end -to-end	Source/binary	Symbolic environment
catchconv	**NO**		**Binary**	**Yes**
KLEE	**NO**		**Source**	**Incomplete**
S^2E	**NO**		**Binary**	**Yes**
Bitblaze	**NO**		**Binary**	**No**
!Exploitable	**NO**		**Binary**	**NO**
Heelan	**Yes**	**Yes**	**Binary**	**Incomplete**
AEG	**Yes**	**Yes**	**Source**	**Incomplete**
MAYHEM	**Yes**	**Yes**	**Binary**	**Incomplete**
CRAX	**Yes**	**Yes**	**Binary**	**Yes**

表格 2 綜合比較

第三章

研究方法

這個章節分成兩部分，第一部分介紹符號執行如何運行在一個建置簡單的符號環境，第二部分說明如何選定一個適當的長度，來當作單一路徑擬真執行的符號輸入長度。

3.1 程式碼的符號執行

考慮程式碼 3 做擬真執行測試，這隻受測程式將資料流從檔案取出接著存到變數 ch，如果要將資料流符號化，只需將程式碼 3 的第四行改程式 4 碼的第四行直將檔案讀取的資料符號化。

```
1.  FILE *fd;
2.  fd = fopen(“file” , “w”);
3.  char ch;
4.  fread(&ch , 1 , 1 fd)
5.  if(ch==’a’)
6.          printf(“ch = a”);
7.  else
8.          printf(“ch !=a”);
```

程式碼 3 符號執行比較組

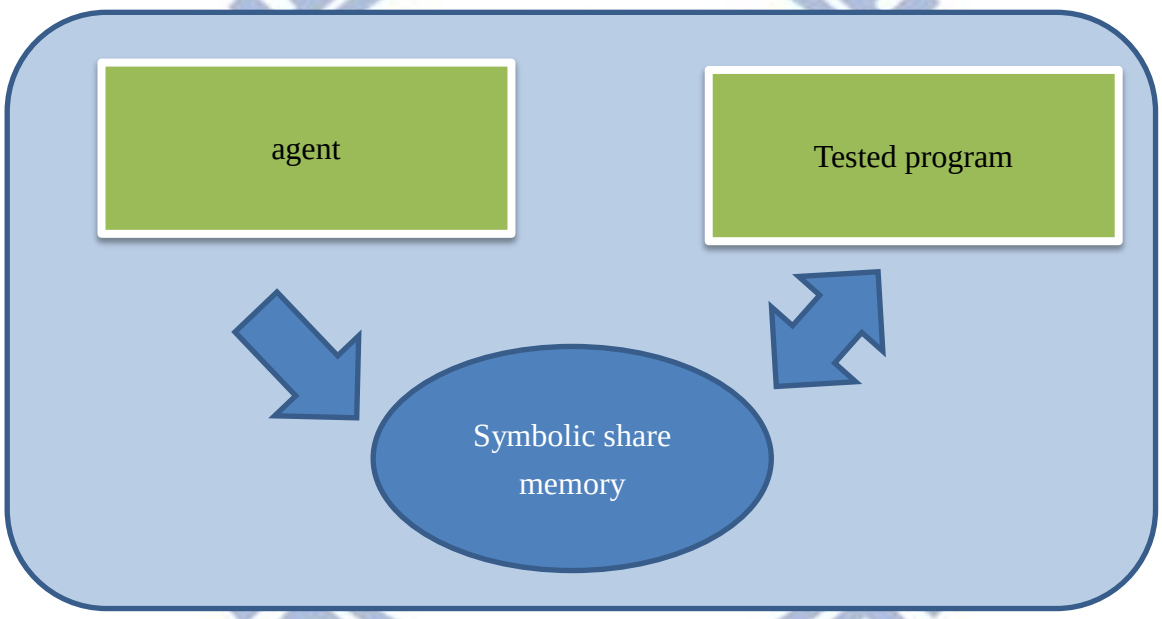
```
1.  FILE *fd;
2.  fd = fopen(“file” , “w”);
3.  char ch;
4.  fread(&ch , 1 , 1 fd)
5.  make_symbolic(&ch, 1);
6.  if(ch==’a’)
7.      printf(“ch = a”);
8.  else
9.      printf(“ch !=a”);
```

程式碼 4 符號執行對照組

然而在執行檔的符號執行中，因為缺乏受測程式原始碼，並無法透過修改程式碼來達成資料流的符號化。

3.2 代理程式與共享符號記憶體

代理程式的工作即為，代替受測程式將資料流符號化，因此代理程式預先建立一塊符號記憶體（symbolic memory），接著將這塊記憶體共享或繼承給在同一台機器上的受測程式如圖表 6，但由於資料流的來源不同，所以共享符號記憶體的方式也不同，以下針對五種不同的資料流來源共享的方式一一做介紹。

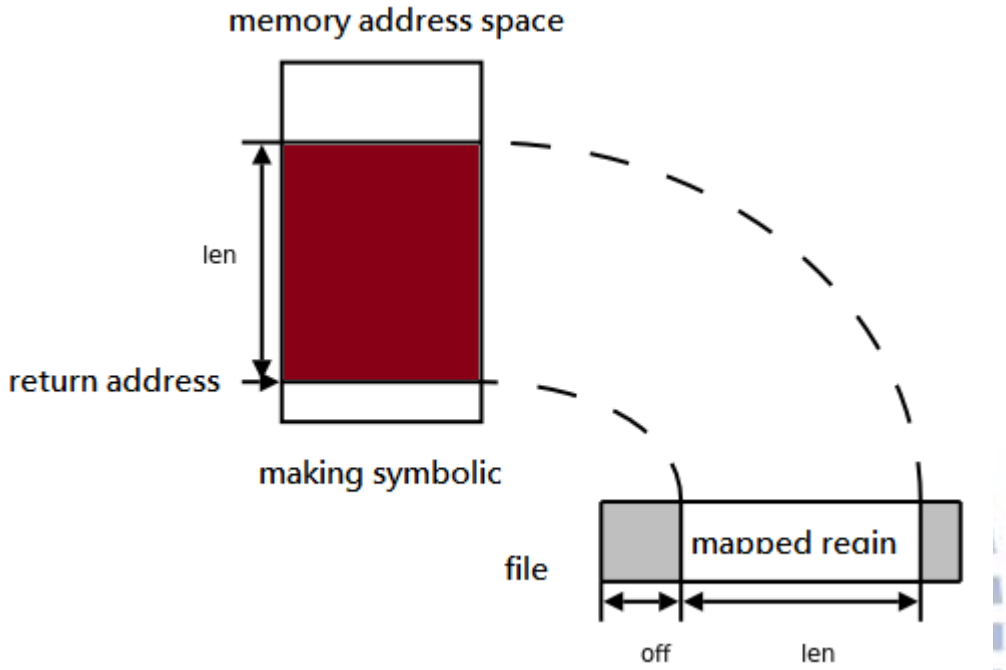


圖表 7 共享符號記憶體

3.3 Symbolic File 建置

建立 Symbolic file 的困難點在於，代理程式並不能只是將一塊共享符號記憶體內容直接寫入到檔案，因為符號記憶體內容一旦進入到檔案系統(file system)，將會失去所有的符號資訊，而被系統強迫轉成一個固定的值(concrete value)，因此這篇論文嘗試讓檔案存取無須涉及檔案系統，而將檔案預先對應到一塊記憶體空間上，如圖表 7，這代表受測程式如果有去讀取這個檔案，其實就是直接對記憶

體操作，而非透過檔案系統來存取資料，接著代理程式再將這一塊對應到檔案的記憶體符號化，完成對受測程式共享符號記憶體的概念。



圖表 8 檔案對應至記憶體中

3.4 Symbolic Socket 建置

在網路程式裡，資料流會在伺服端與客戶端流動，因此 Symbolic socket 分成兩個部分一個是符號資料流由伺服端傳給客戶端，稱作 symbolic server 如圖表 9，另一個則是由客戶端傳給伺服端，如圖表 8，稱為 symbolic client，當建置 symbolic server 時可把伺服端想成是代理程式，而客戶端則是受測程式，因此代理程式首先與客戶端建立連線接著建立一塊符號記憶體並且等待客戶端的要求，一旦符號記憶體資料由代理程式傳到客戶端，客戶端即可開始符號執行，而建置 symbolic client 則相反，代理建立相關連線後主動傳送符號內容給伺服端，則伺服端接收資料後即開始符號執行。

Read symbolic from client	Read symbolic from server

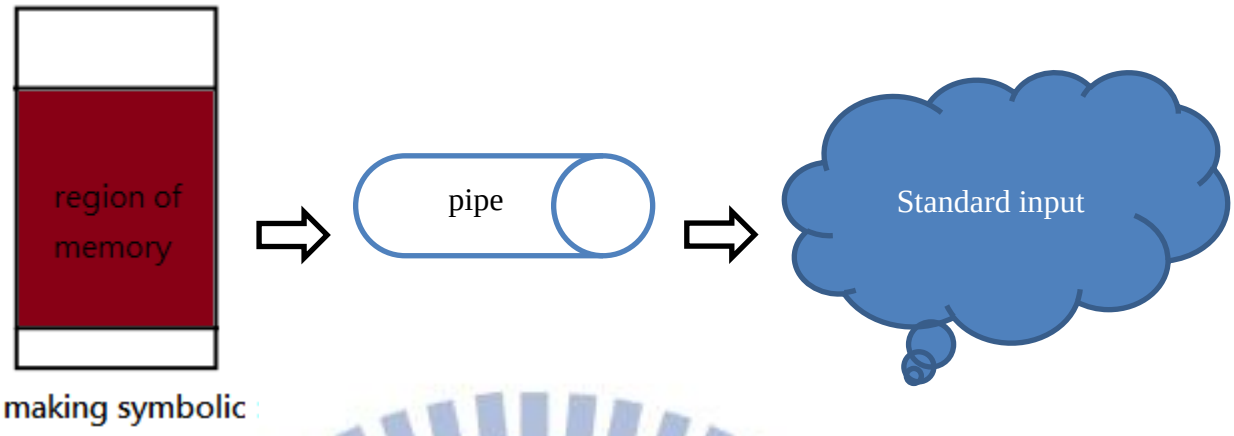
圖表 9 symbolic client　　圖表 10 symbolic server

3.5 Symbolic Environment Variables 建置

代理程式首先向作業系統宣告一個環境變數並且把整塊環境變數符號化，接著代理程式透過函式呼叫”fork”產生新的行程(process)，這支新的行程與原行程擁有相同的符號記憶體，最後新行程透過函式呼叫”execv”啟動受測程式，一旦受測程式讀取符號環境變數，即會開始符號執行。

3-6 Symbolic Standard Input 建置

在 symbolic standard input 裡代理程式首先建立一條 pipe，pipe 是 c 語言所提供的函式呼叫(function call)可以用來更改標準輸入的來源，接著代理程式將 pipe 的一端連接到已經配置好的符號記憶體而另一端則連接標轉輸入，下一步驟，代理程式透過函式呼叫(function call)”fork”來產生一個新的行程(process)，這個行程有著和代理程式相同的記憶體配置，並且標準輸入也同樣被導向到符號記憶體，最後這個行程，在透過函式呼叫”execv”來啟動受測程式，此時受測程式如果從標準輸入讀取資料，將直接對符號記憶體讀取資料並且開始符號執行，而不是從鍵盤接收輸入。圖表 10 為 Symbolic Standard Input 示意圖。



圖表 11 symbolic standard input

3.7 Symbolic Argument Variables 建置

在建置 Symbolic Argument Variables 中，首先代理程式建立一塊共享符號記憶體，接著代理程式再透過函式呼叫(function call) “execl”直接把共享符號記憶體內容當成啟動受測程式的參數，受測程式讀取參數後(實則直接從共享符號記憶體裡讀取資料)後即開始符號執行。

3.8 調適性符號輸入選擇(Adaptive Symbolic Input Selection)

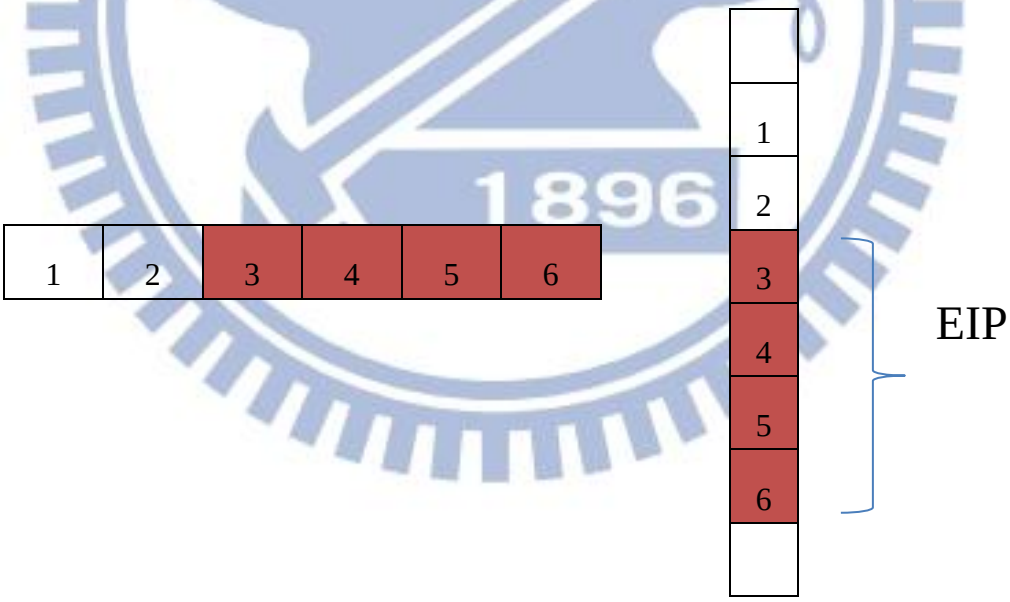
由於 CRAX 在自動脅迫產生(automatic exploit generation)的效率深受符號輸入長度影響，符號長度越長，CRAX 效能就越慢。為了減少符號的輸入長度，本篇論文將符號輸入分成兩部分：關鍵位元組(critical byte)，與其他位元組，所謂關鍵位元組就是當程式在執行一個 type 1crash 的檔案時，EIP 暫存器裡存放的位元組，其他位元組則完全不影響 EIP 暫存器，有這樣的特性，本方法即藉由最小化其他位元組的符號化來達到加速的目的。

3.8.1 符號汙染分析(symbolic taint analysis)

為了找尋 crash 檔案裡的關鍵位元組(critical byte)，本篇論文採取的一個新

的方法稱為符號汙染分析，這個方法將同時運用符號執行(symbolic execution)與污染分析(taint analysis)的特性來做搜尋。

具體的方法是首先透過 crash 檔案來執行單一路徑擬真執行，但是只針對部分的 crash 檔案符號化。這樣的用意是為了縮短符號輸入長度，來加速單一路徑擬真執行，並且在執行過程中，並不記錄任何的分支限制式(branch constraint)，此目的也是避免在維護路徑限制式(path constrain)時花多餘的時間，當執行結束時，如 EIP 暫存器內容為符號變數，就可以根據符號變數紀錄的資訊如圖表 11 來找出 crash 檔案裡對應的位元組，如果 EIP 暫存器沒有被汙染(tainted)，則位移符號輸入四個位元組，並且反覆執行上述的動作，直到搜尋到 crash 檔案裡的關鍵位元組。其演算法為 演算法 1



圖表 12 當 EIP 暫存器被符號變數汙染，可以直接知道來源與目標對應的關係

Q

```
Input V : A input string
      EIP : The value of EIP register
      SIZE : The size of crash file
Output S : A set of critical byte

start <- 0
while start < SIZE
    if SIZE < start + 4 then
        make_symbolic( V(start) , SIZE-start)
        symbolic_execution(V(start))
        i <- start
        while i < SIZE - start
            b<-isSymbolic(EIP(i))
            if b != 0 then
                S.add(EIP(i))
            else
                continue
        i <- i + 1
    else
        make_symbolic( V(start) , 4)
        symbolic_execution(v(start) , 4)
        i <- start
        while i < start + 4
            b<-isSymbolic(EIP(i))
            if b != 0 then
                S.add(EIP(i))
            else
               continue
            i <- i + 1
    start <- start + 4
```

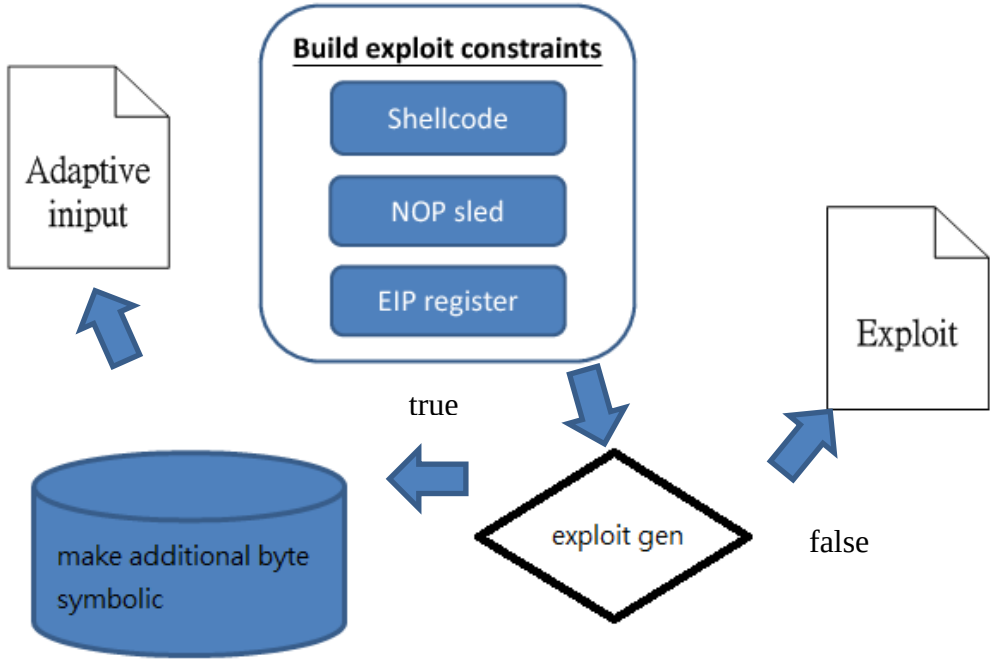
./s

演算法 1 符號汙染分析的演算法

3.8.2 選擇性輸入(input selection)

一個脅迫(exploit)檔案主要包括三部分 shellcode 長度，NOP sled，以及一塊指向 shellcode 的位置，也就是關鍵位元組。因此符號輸入長度至少要包含上述這三項要素的長度，這樣 CRAX 才有足夠的資訊去計算是否可以產生脅迫

(exploit)，如果脅迫產生失敗，則增加符號輸入長度一個位元組直到可以產生脅迫為止。圖表 12 為選擇性輸入完整的流程



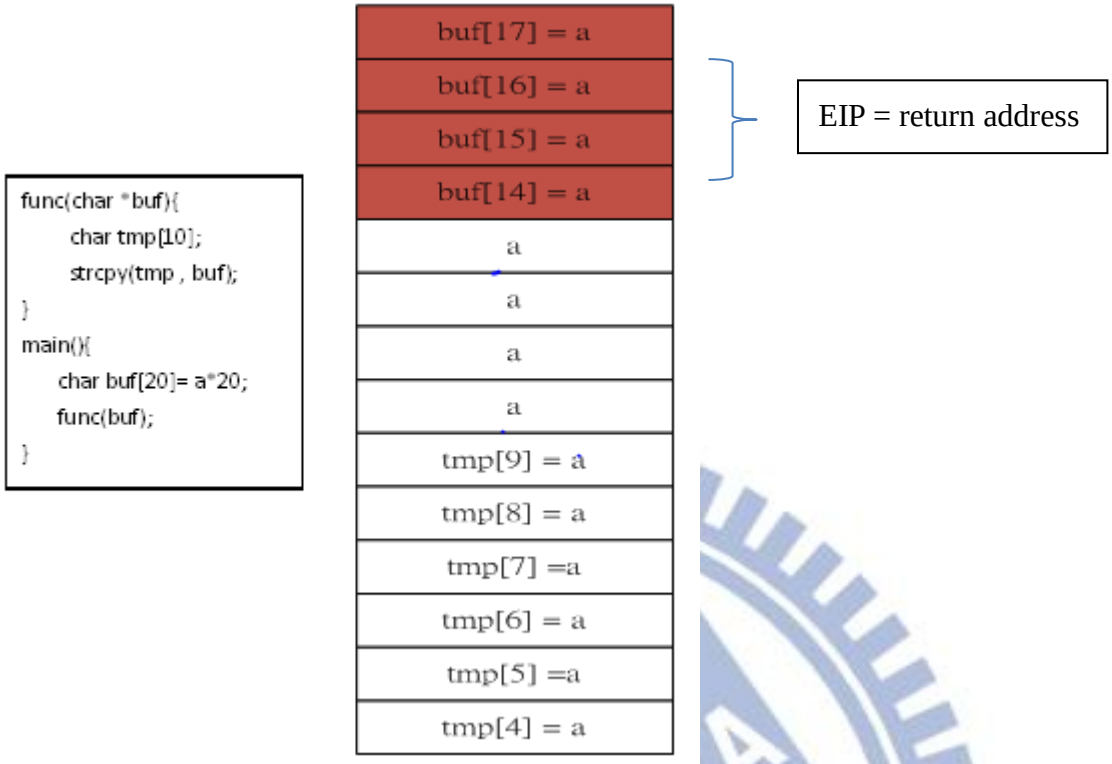
圖表 13 選擇性輸入流程圖

```
Input L : The address of critical byte in input
      I : The address of input string
      Shellcode : The size of shellcode
      NOP : The size of NOP sled
      A :  Additional byte for exploit generation
Output file : exploit file

start <- L
Shellcode <- 30
NOP <- 16
A <- 0
while I < L

    if L - Shellcode - NOP > I then
        make_symbolic(L-Shellcode-NOP , 50+A)
        if file <- exploit_gen(I) then
            break
        else
            L <- L - 1
            A <- A + 1

    else
        make_symbolic(I , L-I)
        if exploit_gen(I) then
            break
        else
            L <- I    //can't not generate exploit
```

演算法 2 選擇性輸入

3.8.3 調適性符號輸入選擇的特殊情況

在 type 1 crash 當中，例如程式碼 4，EIP 暫存器通常直接被 crash 檔案裡的後半段覆蓋到(如圖表 13，EIP 被 buf[14]~buf[17]蓋到)，因此可以直接假設關鍵位元組在 crash 的後半段，而跳過符號汙染分析這個步驟直接進行選擇性輸入來達到更加的效率。



圖表 14 堆疊溢位

程式碼 5 TYPE 1 crash

第四章

實作

本章首先介紹符號環境建置的實作方法，五種符號環境 symbolic file，symbolic socket，symbolic standard input，symbolic environment variable，symbolic argument variable 的代理程式直接實作在 guest OS 上並透過 S^2E 提供的 API 與 host OS 端做溝通。

適性符號輸入選擇需要動態的調整符號輸入的位置與長度，因此在實作符號汙染分析與選擇性輸入前，還必須先了解 S^2E 的記憶體架構。

4.1 S^2E API 介紹

s2e_make_symbolic(address , size , name);

這個 API 會在記憶體裡從 address 開始分配一塊大小為 size 的符號記憶體 (symbolic memory)，並且代號為 name。

s2e_kill_state(0, "program terminated");

S^2E 對於擬真執行中每一條程式執行路徑都以一個 state 代表，而這個 API 的作用是停止當前 state。

4.2 Symbolic File

代理程式將透過函式呼叫”mmap”將真實檔案映射到記憶體裡，第 11 行 mmap 回傳檔案在記憶體裡的起始位置。第 13 行根據起始位置以及檔案大小，將在記憶體裡的整個檔案符號化。第 12 行是一個很重要的步驟，因為如果沒有預先存取將被映射的記憶體，檔案就不會被搬移到目的地，第 15 行也就會配置失敗。第 14 行無論有無產生新的行程(process)都不影響符號記憶體被其他程式共享，因為檔案本就可以同時被多支行程共用，因此映射到記憶體後，任何程式同樣也可以透過記憶體來存取檔案。而這也是共享符號記憶體的目的

```
1.  #include <sys/mman.h>
2.  int main(){
3.
4.      int fd;
5.      pid_t pid;
6.      int i;
7.      char *ptr;
8.      fd = open("./test",O_RDWR);
9.      struct stat b;
10.     fstat(fd , &b);
11.     ptr=mmap(0,b.st_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
12.     ptr[0] = 'a';
13.     s2e_make_symbolic(ptr,b.st_size, "buf" );
14.     if((pid = fork())==0){
15.         execv("./tested program", NULL);
16.     }else{
17.         wait();
18.         close(fd);
19.         s2e_kill_state(0, "program terminated");
20.     }
21.     return 0;
22. }
```

程式碼 6 symbolic file

4.3 Symbolic socket

4.3.1 Symbolic Server

首先代理程式建立一條連線以供客戶端連線，第 14 行建立一塊符號記憶體，第 15 行可以看出這個範例伺服端僅簡單的將符號記憶體內容傳給受測程式，如果兩端的行為更複雜則需實作更多的程式碼。

4.3.2 Symbolic Client

代理程式，首先與伺服端建立連線，建立一塊符號記憶體，即可將符號記憶體內容當成資料傳送給伺服端，程式碼 7 與 symbolic server 一樣僅處理簡單的網路程式行為，如果溝通的行為更複雜，必須實作而外的功能。

```
1.  main(){
2.
3.      ...
4.      set up
5.      ...
6.
7.      bind(...);
8.      listen(...);
9.      char ch;
10.     s2e_make_symbolic(ch, 1 , "ch" , 0);
11.    while(1){
12.        client_len = sizeof(client_address);
13.        client_sockfd = accept(...)
14.
15.        write(client_sockfd , &ch , 1);
16.        close(client_sockfd);
17.    }
18.    return 0;
19. }
```

程式碼 7　symbolic server

```
1.  main(){
2.
3.
4.      ...
5.      set up
6.      ...             .
7.
8.      sockfd = socket(...)
9.      char buf[4]="aaaa";
10.
11.     connect (...);
12.     s2e_make_symbolic(buf , 4 , "buf" , 0);
13.     write(sockfd , &buf[0] , 1);
14.     close(sockfd);
15.     return 0;
16.
17. }
```

程式碼 8　symbolic client

4.4　Symbolic Environment Variables

在程式碼 8 中代理程式第 4 行透過呼叫 setenv 向作業系統宣告一個新的環境變數，第 5 行透過 getenv 取得環境變數在記憶體裡的起始位置，接著依據環境變數的大小，以及在記憶體的位置，建置一塊環境變數的符號記憶體，這邊額外討論 s2e_make_symbolic 的時機，s2e_make_symbolic 可以分成放在 setenv 之前或者是 getenv 之後這兩種策略，第一種的符號執行可以記錄更完整的路徑限制式，例如環境變數有哪些字元是不可被使用的，都會被詳實記錄，但是缺點就是花額外的時間對 setenv 做符號執行，而第二種策略因為是在 getenv 之後才分配符號記憶體，所以沒有策略一的問題，但是路徑限制式就相對不完全。8 至 9 行代理程式複製新的行程(process)並執行受測程式，使受測程式將有著和代理程式一樣的符號環境變數記憶體。

```
1.  main(int argc, char **argv){
2.
3.          char input[10] = “aaaaaaaaaa”;
4.          setenv(“env var”, &input, 1);
5.          char *ptr = getenv(“env var”);
6.          s2e_make_symbolic(ptr, 10, "env");
7.          int pid = fork();
8.          if(pid == 0 ) {
9.              execl(program, program, NULL);
10.         } else {
11.             wait();
12.             s2e_kill_state(0, " program terminated ");
13.         }
14.         return 0;
15. }
```

程式碼 9 Symbolic Environment Variables

4.5 Symbolic Standard Input

函式呼叫(function call)pipe 有兩個端點，輸入端與輸出端，行程可以從輸出端讀取輸入端的資料，第 7 行將已經建置好的符號變數寫入輸入端，而第 9 行 dup2 可以設定標準輸入可以從 pipe 的輸出端讀取資料。這些 pipe 的設定都會透過 fork 複製到新的行程，當新的程式執行受測程時，受測程式即可透過標準輸入來讀取符號記憶體裡的內容。

```c
main(){
    int pipe_fd[2];
    pid_t pid;
    char buf[FILE_SIZE];
    memset(buf , 0 , sizeof(buf));
    s2e_make_symbolic(buf,FILE_SIZE, "buf", );
    write(pipe_fd[1] , buf , FILE_SIZE);
    if((pid = fork())==0){
        dup2(pipe_fd[0] , 0);
        close(pipe_fd[0]);
        close(pipe_fd[1]);
        execv("./stdin2", NULL);
    }else{
        close(pipe_fd[0]);
        close(pipe_fd[1]);
        wait();
        s2e_kill_state(0, "program terminated");
    }
    return 0;
}
```

程式碼 10 symbolic standard input

4.6 Symbolic Argument Variables

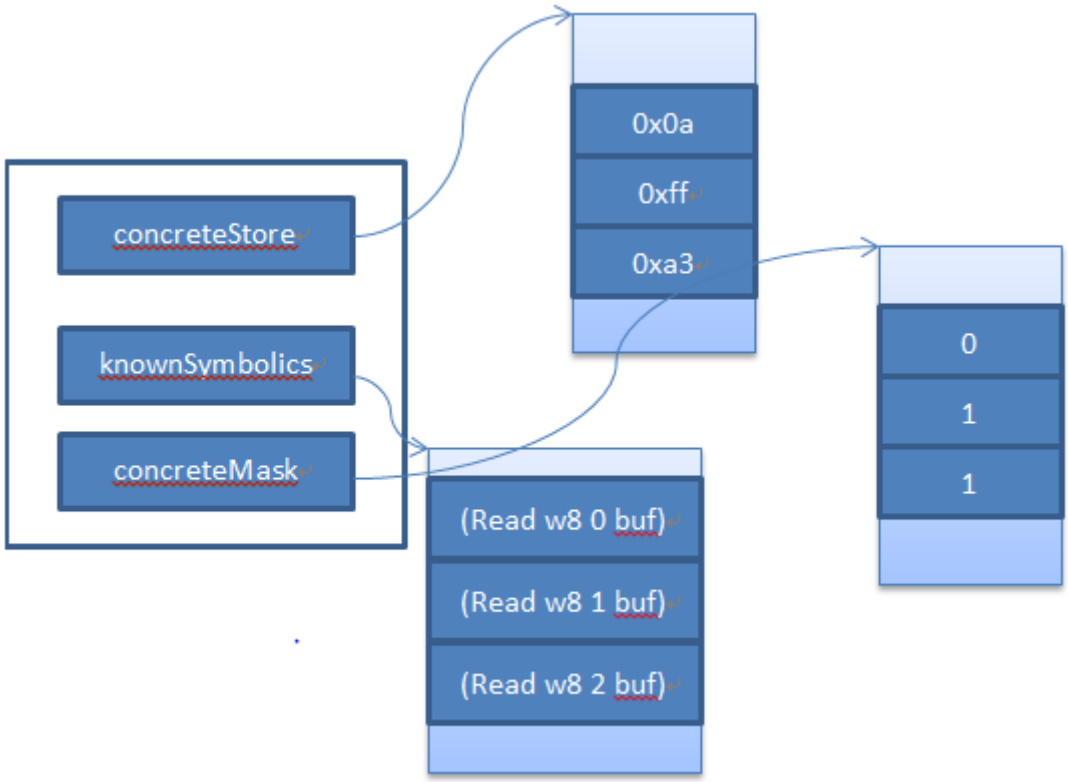
Symbolic argument variable 的代理程式是所有符號環境建置實作最簡單的一支程式，代理程式只需要預先建立一塊符號記憶體，再透過函式呼叫”execl:將符號記憶體傳給受測程式使用，即可完成。

```
1.
2. main(void){
3.
4.     char str[2]="aa";
5.     s2e_make_symbolic(str, 2, "str");
6.     pid_t pid;
7.     pid = fork();
8.     if(pid ==0){
9.         execl(“program" , “program" , str , NULL);
10.    }else{
11.        wait();
12.        s2e_kill_state(0, "program terminated");
13.    }
14.    return 0;
15. }
```

程式碼 11 Symbolic Argument Variables

4.7 S²E 的記憶體架構

S²E 用軟體的方式來管理 guest 作業系統的記憶體資料。S²E 建立了一個資料結構：ObjectState 物件，裡面有三個欄位，分別是 concreteStore，knownSymbolic，以及 bitarray，bitarray 用來標示一段記憶體裡那些位元組為符號變數，那些位元組則是常數，而 knownSymbolic 則指向 host 記憶體裡存放符號變數的位置，concreteStore 則指向 host 記憶體裡存放常數的位置。因此 guest 程式在讀取符號變數時 S²E 就透過 ObjectState，找出符號變數裡的表示式(express)然後回傳給 guest。物件 ObjectState 的資料結構可以參考圖表 14



圖表 15 物件 ObjectState 資料結構

4.8 符號汙染分析

本篇論文透過 S^2E 提共的函式，啟動多支行程來早尋關鍵位元組(critical byte)並透過修改 ObjectStatex 來對每一隻行程執行的符號輸入位置與長度做修改，並且每一隻行程的符號輸入長度為四組位元組。

第五章

結果與實驗

本章節分成四部分，第一部份討論新版本 CRAX 採取調適性符號輸入選擇的策略後效能的改進，第二部分則比較 AEG 與新 CRAX 在自動脅迫產生(automatic exploit generation)上的速率，第三部分則討論新 CRAX 與 MAYHEN 的測試數據。第四部份針對這些實驗將給出一個總結。

5.1 實驗環境

本論文所有實驗皆運行在虛擬機器上其環境為：

實體機器：INTEL XEON E5620 2.4GHz

記憶體大小：40GB

Host OS：Ubuntu 10.10 64-bit

Guest OS：Debian 5.0 32-bit

5.2 新舊 CRAX 效能比較

表格 3 為舊 CRAX 與新 CRAX 的比較，其中第二行以及第四行為符號輸入的長度。新版本比較版本平均加速 25 倍，最大加速 142 倍(ncompress)最小加速 2.1 倍（xmail），並且新版 CRAX 除了 xmail 以外每一隻程式的執行時間皆小於 5 秒

而舊版 CRAX 執行時間小於 10 秒的有 4 隻程式，介於 10 秒到一分鐘內的有 7 隻程式，大於一分鐘的有 5 隻程式

Program	Input length	CRAX time	Adaptive input length	CRAX Time (Adaptive)	Speedup
aeon	550	32.0	50	2.6	12.3x
iwconfig	85	3.6	50	0.7	5.1x
glftpd	300	8.0	50	0.5	16x
ncompress	1050	99.4	50	0.7	142x
htget (arg)	276	35.5	50	2.9	12.2x
htget(env .var)	180	5.1	50	1.17	4.3x
expect (HOME)	300	29.4	50	2.7	10.8x
expect(DOTDIR)	300	29.3	50	3.56	8.2x
rsync	201	9.9	50	2.7	3.6x
acon	1300	32.0	50	2.7	11.8x
gif2png	1080	154.7	50	1.69	91.5x
hoslink	1050	103.9	50	2.4	42.9x
exim	304	122.3	50	4.3	28.4x
aspell	300	14.5	50	1.7	8.5x
xserver	104	14.4	50	2.5	41.6x
xmail	307	371.7	50	171.0	2.1x

表格 3 新舊 CRAX 比較

5.3 與 AEG 標竿程式的比較

表格 4 中新 CRAX 平均比 AEG 測試時間快 16 倍，最大快 69.5 倍(expect_HOME)。而在 htget_env 上速度則是持平，另外還可以發現 AEG 測試的時間如果大於 10，則新 CRAX 至少加速 5 至 60 倍，而 xmail 為 AEG 花最多時間測試的程式，約為 21 分鐘，而新 CRAX 則需要約 3 分鐘的時間即可產生 xmail 的脅迫(exploit)。值得一提的是，AEG 在測試程式的時候需要受測程式的原始碼來幫助分析，而 CRAX 則是直接測試執行檔，一般來說擁有受測程式的原始碼將更使分析程式更容易，因此新版 CRAX 能夠在相對缺乏額外的受測程式資訊下，能夠有此結果，顯示調適性符號輸入選擇有著關鍵的影響力。

Program	Input length	AEG time	Adaptive input length	CRAX (Adaptive)	Speedup
aeon	550	3.8	50	2.6	1.5x
iwconfig	85	1.5	50	0.7	2.1x
glftpd	300	2.3	50	0.5	4.6x
ncompress	1050	12.3	50	0.7	17.5x
htget (arg)	276	57.2	50	2.9	19.72x
htget(env .var)	180	1.2	50	1.17	1.x
expect (HOME)	300	187.6	50	2.7	69.5x
expect(DOTDIR)	300	187.7	50	3.56	52.7x
rsync	201	19.7	50	2.7	7.3x
exim	304	33.8	50	4.3	7.8x
aspell	300	15.2	50	1.7	8.9x

xserver	104	31.9	50	2.5	12.7x
xmail	307	1276.0	50	171.0	7.5x

表格 4　AEG 標竿程式

5.4 與 MAYHEM 的比較

這個實驗從 MAYHEM 挑選出１１隻程式來做效能的比較，其中有７隻受測程式是原 AEG 的標竿程式，表格可以計算出新版 CRAX 平均比 MAYHEM 快上 13 倍，而在 MAYHEM 最長的測試時間為 362 秒(Mbse-bbs)，新版 CRAX 則為 26.9 秒。

Program	Input length	Mayhem time	Adaptive input length	CRAX (Adaptive)	Speedup
aeon	550	10.0	50	2.6	38.4x
aspell	85	82.0	50	1.7	48.2x
glftpd	300	4.0	50	0.5	8x
Htget	350	7.0	50	1.17	5.9x
Iwconfig	400	2.0	50	0.7	2.9x
ncompress	1400	11.0	50	0.7	15.7x
rsync	100	8.0	50	2.7	3x
Mbse-bbs	4200	362.0	50	26.9	13.4x
psutils	300	46.0	50	25.4	1.8x
htpasswd	400	4.0	50	0.4	10X
Squirrel mail	150	2.0	50	0.9	2.2X

表格 5 與 MAYHEM 比較

5.5 測試大型程式

新版 CRAX 在使用調適性符號輸入選擇方法在產生大型程式的脅迫(exploit) 有著明顯的突破,表格 6 為 unrar 以及 mplayer 自動產生脅迫的時間,可看出 unrar 平均加速 293 倍，Mplayer 平均加速 41 倍，

Program	**Input Length**	**Explore Time**	**Exploit Gen. Time**	**Total time**	Adaptive input length	**Explore Time (Adaptive)**	**Exploit Gen. Time (Adaptive)**	**Total time (Adaptive)**
Unrar	5000	1388.5	2569.8	3958.3	120	11.7(118x)	1.8(1427x)	13.5(293x)
Mplayer (Linux)	145	145.8	3.56	149	54	3.3(44x)	0.3(11x)	3.6(41x)

表格 6 大型程式測試數據

5.6 小結

CRAX 在使用調適性符號輸入選擇明顯速度上升很多，主要原因在於符號變數長度大量縮短,但是還沒辦法根據這些實驗的數據推論出減少多少符號長度能夠帶來多少的加速，例如在新舊版 CRAX 的比較中 aeon 的符號輸入長度減少 26 倍後效能加速 11.8 倍，但 gif2png 符號長度減少 21 倍後效能卻加速了 91 倍，但可以確定的是，舊版 CRAX，AEG，以及 MAYHEM 執行越沒效率的程式，調適性符號輸入選擇的效果就越明顯。

第六章

結論與未來研究

6.1 結論

本篇論文，提供了一個簡單的方法在 S^2E 平台上建立符號環境，這個方法不需用修改任何的函式呼叫(function call)，以及系統呼叫(system call)，因此任何人參照這篇論文的實作，可以馬上重建符號環境而不需要花費大量時間與人力，來達到執行檔的黑箱測試。

這邊論文提出的第二個方法：調適性符號輸入選擇，這個方法大大改善了 CRAX 在自動脅迫產生(automatic exploit generation)的效率，也增加了 CRAX 在大程式測試的可行性，而在軟體安全的議題上，面對大量的軟體漏洞，CRAX 也更有能力去判別那些漏洞是最具威脅性，必須優先修補，以減少軟體被惡意攻擊的機會。

6.2 未來研究

6.2.1 符號環境的改進

本論文所提供的符號環境建置方法，主要針對軟體在文字介面(command line)環境下，所有可能了資料流來源，都能符號化，但是在圖形介面的情況下，使用者還可以透過滑鼠來產生不同的事件當作軟體的輸入(input)，為了模擬使用者的行為來進行圖形介面的軟體測試，可以將使用者透過滑鼠所產生的事件符號化，來達到符號測試的目的。另一方面在測試大型的網路程式，伺服端與客戶端，的通訊行行為，沒辦法透過簡單的實作一隻代替程式來模擬，更完善的策略是藉由透過改寫現有的伺服端與客戶端程式，來增加符號執行(symbolic execution)在複雜的網路溝通行為上測試的可行性。

6.2.2 調適性符號輸入選擇的改進

調適性符號輸入選擇雖然在 type 1 crash 的自動脅迫產生(automatic exploit generation)上有個明顯的加速效果，但是並沒有實際運用在測試有 type 2 crash 安全漏洞的軟體，因此缺乏在這種情況的評估。而另一方面在實作選擇性輸入(input selection)上如果演算 2 第一次迴圈無法產生出脅迫時，必須手動執行新的單一路徑擬真執行。

參考文獻

[1] Schwartz, E. J., Avgerinos, T., & Brumley, D. (2010). All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In Ieee (Ed.), *2010 IEEE Symposium on Security and Privacy* (pp. 317–331).

[2] Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C. S., Sen, K., Tillmann, N., et al. (2011). Symbolic execution for software testing in practice: preliminary assessment. In *Software Engineering (ICSE), 2011 33rd International Conference on* (pp. 1066–1071).

[3] King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM*, *19*(7), 385–394.

[4] Sen, K. (2007). Concolic testing. In Acm (Ed.), *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (pp. 571–572).

[5] Cadar, C., Dunbar, D., & Engler, D. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In USENIX Association (Ed.), *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (pp. 209–224).

[6] Avgerinos, T., Cha, S. K., Hao, B. L. T., & Brumley, D. (2011). AEG: Automatic Exploit Generation. In *Network and Distributed System Security Symposium*.

[7] Cha, A. R. S. K., Avgerinos, T., & Brumley, D. (2012). Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy* (Vol. 2, pp. 5–3).

[8] Heelan, S. (2009). *Automatic generation of control flow hijacking exploits for software vulnerabilities*. Ph.D. thesis, University of Oxford, .

[9] Molnar, D. A., & Wagner, D. (2007). *Catchconv: Symbolic execution and run-time type inference for integer conversion errors*. EECS Department, University of California, Berkeley.

[10] Chipounov, V., Kuznetsov, V., & Candea, G. (2012). The S2E Platform: Design, Implementation, and Applications. *ACM Trans.Comput.Syst.*, *30*(1), 1–49

[11] Miller, C., Caballero, J., Johnson, N. M., Kang, M. G., McCamant, S., Poosankam, P., et al. (2010). Crash analysis with BitBlaze. *at BlackHat USA*,

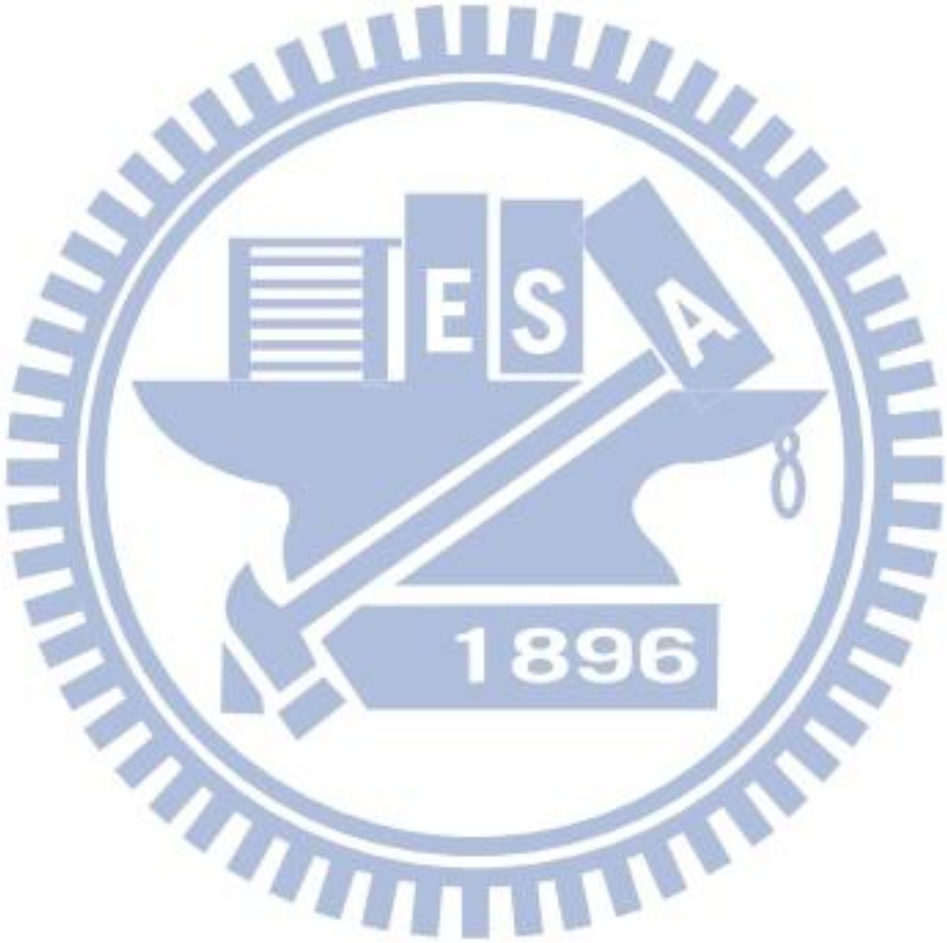
[12] !exploitable crash analyzer. *http://msecdbg.codeplex.com/*

[13] Shih-Kun Huang, Po-Yen Huang, Min-Hsiang Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong (2012), CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations,

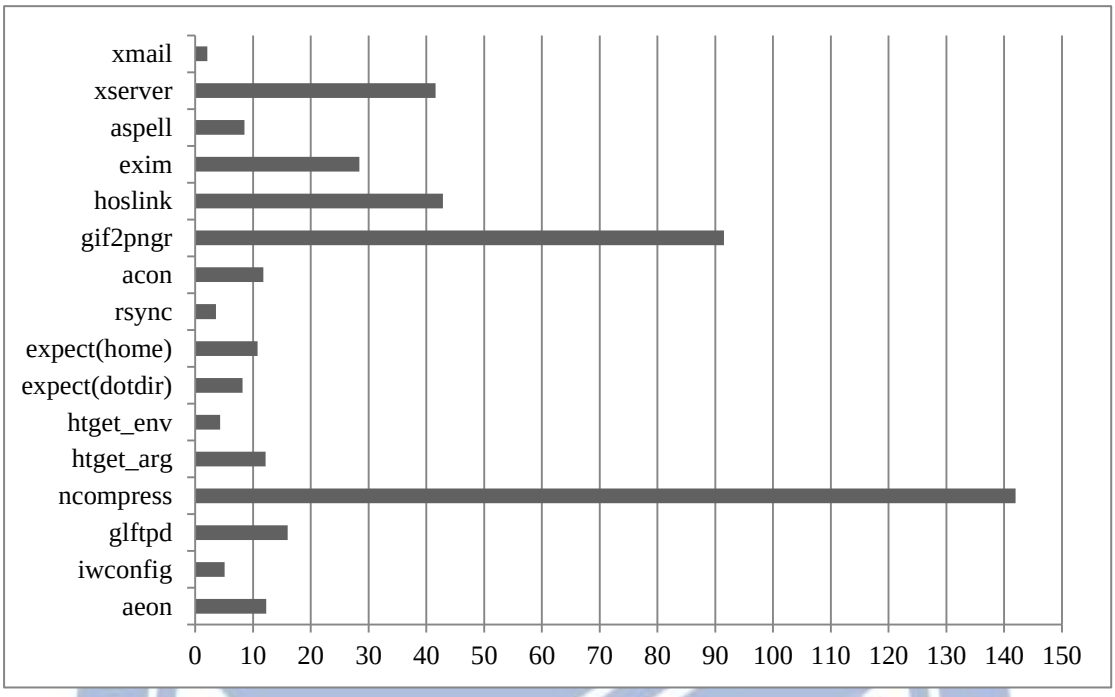
National Chiao Tung University, Taiwan

[14] Lattner, C., & Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (75). Cgo â€™04. Washington, DC, USA: IEEE Computer Society.

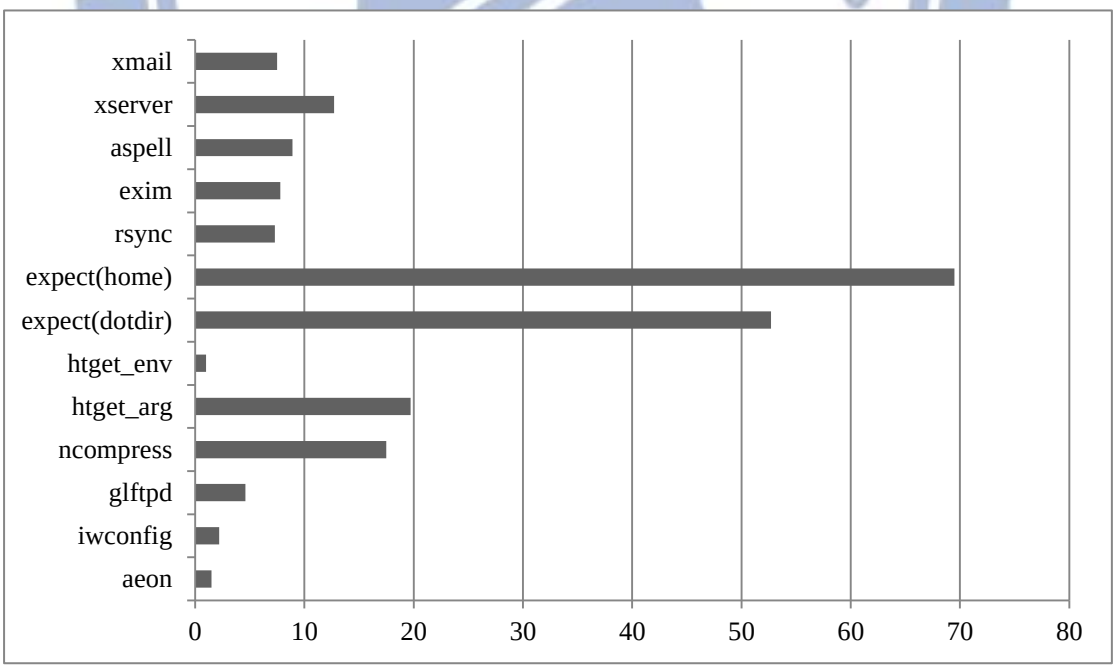
[15] Bellard, F. (2005). QEMU, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (p. 41). Atec â€™05. Berkeley, CA, USA: USENIX Association.



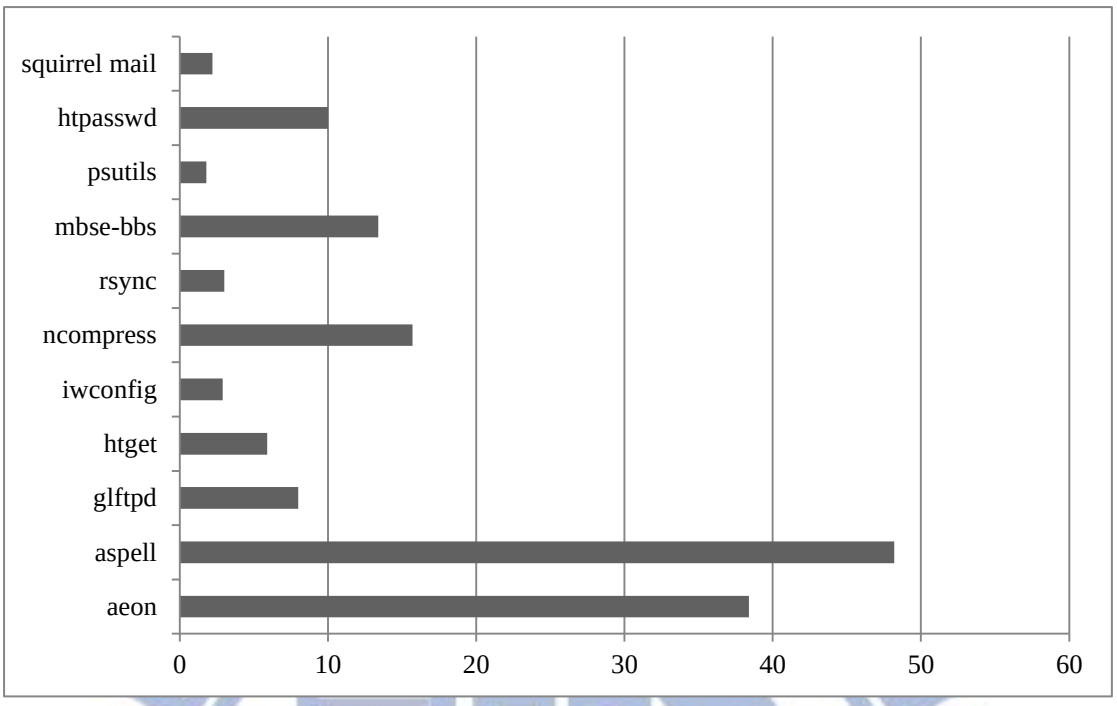
附錄 A 長條圖數據



圖表 16 新舊 CRAX 長條圖數據



圖表 17 使用長條圖與 AEG



圖表 18 使用長條圖與 MAYGEM 比較