

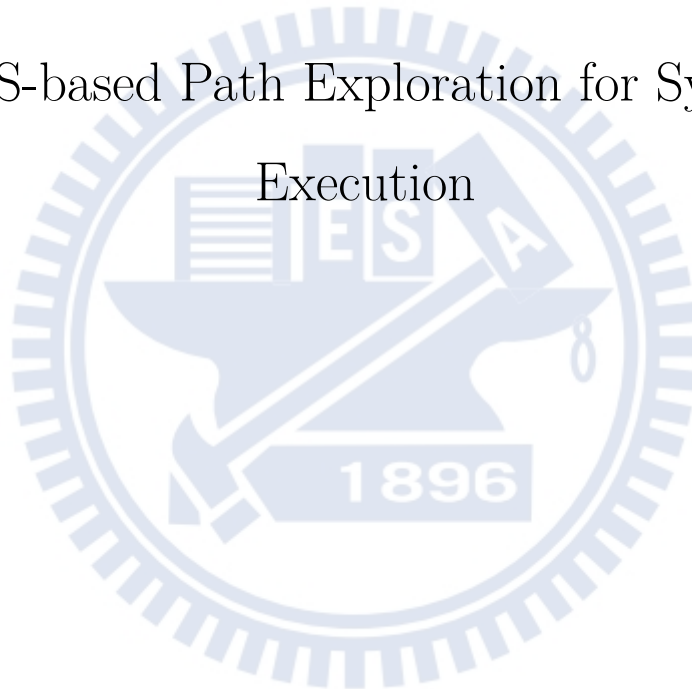
國立交通大學

資訊科學與工程研究所

碩士論文

基於蒙地卡羅樹搜尋法之路徑探索於符號執行

MCTS-based Path Exploration for Symbolic
Execution



研 究 生：葉家郡

指 導 教 授：黃世昆 教授

中華民國 105 年 7 月

基於蒙地卡羅樹搜尋法之路徑探索於符號執行
MCTS-based Path Exploration for Symbolic
Execution

研究生：葉家郡

Student：Jia-Jun Yeh

指導教授：黃世昆

Advisor：Shih-Kun Huang

國立交通大學
資訊科學與工程研究所
碩士論文

A Thesis Submitted to Institute of Computer Science and
Engineering College of Computer Science National Chiao Tung
University in Partial Fulfillment of the Requirements for the
Degree of Master in Computer and Information Science

July 2016

Jia-Jun Yeh, Taiwan

中華民國 105 年 7 月

基於蒙地卡羅樹搜尋法之路徑探索於符號執行

學生：葉家郡
指導教授：黃世昆 教授

國立交通大學資訊科學與工程研究所碩士班

摘 要

在程式自動化測試與分析上，符號執行 (symbolic execution) 是目前經常被使用的一種方法，由於符號執行會紀錄並模擬出程式執行時的所有可能路徑，其數量會以指數的數量級成長，最終耗盡所有運算資源，這個問題被稱為路徑爆炸問題 (path explosion problem)；因此我們需要在有限的資源內採取某些策略來優先計算較有價值的路徑，在本篇論文中我們提出使用以蒙地卡羅搜尋樹為基礎的搜尋策略來解決這個問題，並比較它與其他傳統策略如深度優先搜尋 (DFS)、廣度優先搜尋 (BFS) 的效率。

關鍵字：Monte Carlo Tree Search(MCTS), Upper Confidence Bounds for Trees (UCT), symbolic execution

MCTS-based Path Exploration for Symbolic Execution

Student : Jia-Jun Yeh

Advisor : Dr. Shih-Kun Huang

Institute of Computer Science and Engineering
National Chiao Tung University

ABSTRACT

Symbolic execution is a technology that is often used today in program automation testing and analysis. Since symbol execution traces and simulates all possible paths when a program executes, its number grows exponentially. This problem is called the path explosion problem. Therefore, we need to take some strategies within the limited resources to give priority to the more valuable path. In this paper, we propose to use The Carlo search tree-based search strategy solves this problem and compares it with other classic strategies such as depth-first search (DFS) and breadth-first search (BFS).

Keywords : Monte Carlo Tree Search(MCTS), Upper Confidence Bounds for Trees (UCT), symbolic execution

Contents

1	Introduction	1
2	Background	3
2.1	Symbolic execution	3
2.2	Monte Carlo tree search	4
3	Method	6
3.1	Motivation example	6
3.2	proposed algorithm	8
3.3	algorithm explanation	10
4	Implementation	12
4.1	Angr’s architecture	12
4.2	Exploration techniques	13
4.3	Applied MCTS into Angr	16
5	Result and Evaluation	18
5.1	Environment	18
5.2	演算法與其他方法的比較	19
5.3	MCTS 的評估函式效能測量	21
5.4	長時間執行的效率比較	21
5.5	Discussions	27
6	Related Work	29
6.1	Path Selection Problem	29
6.2	Search-based test generation	30
6.3	MCTS and Game AI	30

6.4	MCTS in AlphaGo	30
7	Conclusions	32
7.1	conclusion	32
7.2	further work	33
Appendix A	實驗數據	38



List of Figures

1	Monte Carlo tree search in AlphaGo Source:D Silver et al. Nature 529, 484–489 (2016) doi:10.1038/nature16961	2
2	symbolic execution 流程	4
3	Monte Carlo Tree Search	5
4	不同策略的樹走訪順序	6
5	Tree of the example code	7
6	MCTS 執行過程	11
7	Example script to run symbolic execution in Angr	13
8	partial code of step in Angr	14
9	partial code of one step in Angr	15
10	partial code of exploration technique(DFS) in Angr	15
11	MCTS of pre step in Angr	16
12	MCTS of post step in Angr	17
13	block coverage in small test	20
14	efficienct in small test	20
15	large test for cpp-markdown	22
16	large test for cp	23
17	large test for echo	23
18	large test for gif2png	24
19	large test for hostname	24
20	large test for ls	25
21	large test for mkdir	25
22	large test for ps	26

23 large test for readelf 26

24 large test for touch 27



List of Tables

1	Target Program's name and version	18
2	Comparison of the efficiency with different strategies	19
3	Comparison of the efficiency with random selection	21
4	cpp-markdown & cp 長時間執行數據	38
5	echo & gif2png 長時間執行數據	39
6	hostname & ls 長時間執行數據	39
7	mkdir & ps 長時間執行數據	40
8	readelf & touch 長時間執行數據	40

Chapter 1

Introduction

在現今的軟體開發中，程式碼數量動輒數萬行，系統的複雜度也越來越高，傳統驗證程式正確性的方式如 unit testing、code review 等等... 受限於人工而相當有限。在硬體計算能力突飛猛進的現代，自動化測試的方式又逐漸成為顯學，如 fuzzing、symbolic execution... 能夠自動尋找程式中可能的漏洞，其中 symbolic execution 是一種模擬執行的方法，它將程式的使用者輸入視為符號，並把程式執行過程和分支條件轉換為限制式，藉由求解限制式來獲得欲執行該路徑所需的使用者輸入為何；由於 symbolic execution 在遇到分支時會複製出一條新的路徑，兩條路徑分別探索執行 if 時和執行 else 時的狀態，因此路徑數量會以指數的數量級成長，記錄並運算這些路徑需要花費巨大的電腦資源，這個問題被稱為路徑爆炸問題 (path explosion problem)，這篇論文欲以蒙地卡羅樹搜尋演算法來找出探索價值較高的路徑，使得 symbolic execution 能使用較短的時間內獲得更高的程式執行覆蓋率。

蒙地卡羅樹搜尋 (Monte Carlo tree search) 被廣泛的運用在遊戲人工智慧中，例如西洋棋、黑白棋、圍棋等等的棋盤遊戲，在 2016 年 AlphaGO [1](一個結合蒙地卡羅樹搜尋和深度學習的圍棋 AI 程式) 擊敗世界棋王後，更是一度引起相當多的討論。雖大眾關注的焦點多在人工智慧發展的進步、類神經網路應用到遊戲 AI 上時帶來的優異成績，卻鮮少提及與類神經網路演算法相互合作的重要元件—樹搜尋策略，Figure 1 為 [1] 中的樹搜尋策略演算法架構，在不同的演算法步驟下使用個別的類神經網路來引導 MCTS 做出選擇，如 b. Expansion 用的是 supervised learning policy network、c. Evaluation 分別使用了 value network 和 rollout policy network 的結果相加來評估盤面價值。MCTS 主要的精神在於：對一棵空間狀態樹，如何選擇節點，並以模擬的方式推估該節點的可能發展狀況和價值，藉由不斷的重複此動作來逼近最佳解。

現今演算法的發展上已經逐漸地從利用人工設計的規則條件進展到讓機器自我建立

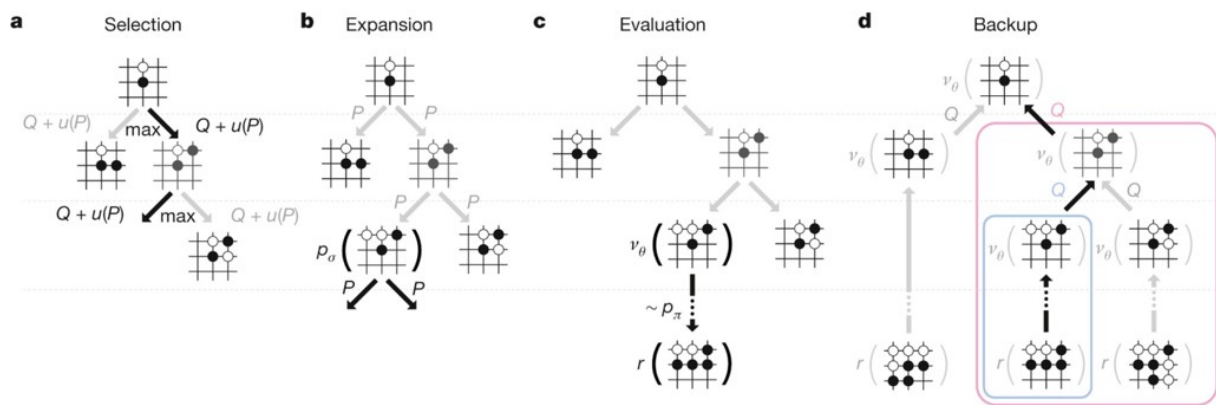


Figure 1: Monte Carlo tree search in AlphaGo Source:D Silver et al. Nature 529, 484–489 (2016) doi:10.1038/nature16961

規則的學習式機制，在各種領域如圖像辨識、資料探勘、自然語言處理、手寫文字識別、遊戲 AI、機器人都有著蓬勃的發展，在這篇論文中預期以 MCTS 的角度切入，導入 symbolic execution 的路徑選擇演算法中，並比較其與傳統策略的效能差異。並希望在以 MCTS 為架構的演算法被建立後，在其之上加入機器學習的方法，如類神經網路、強化學習 (Reinforcement learning)...，使得效能進一步增加，最後達到在漏洞挖掘上超越人類手工尋找的地步。

Chapter 2

Background

在這個章節將簡要介紹 symbolic execution 與其遇到的問題，還有蒙地卡羅樹搜尋演算法的計算流程。

2.1 Symbolic execution

在本篇論文中所提及的 symbolic execution 屬於 Dynamic symbolic execution，首先由 K. Sen[2] 提出，和基於其想法實作的程式 DART[3]、CUTE[4]，而其近年又分為兩種類型：需要程式原始碼的 code-based symbolic execution 如 KLEE[5]，和不須程式碼而直接分析程式執行檔的 binary-based symbolic execution 如 S2E[6]、Mayhem[7]。symbolic execution engine 在分析程式時會將其載入並先轉換為 intermediate representation (IR)，接著會將使用者輸入 (例如標準輸入、檔案、命令列參數) 標記為 symbolic 變數，接著模擬程式的執行過程，將執行過程中遇到的程式碼轉換為數學邏輯限制式的形式，當在模擬的過程中遇到分支條件時，便複製出一條新的路徑，分別追蹤該分支條件為真和為假時的情況；當模擬執行結束時，把先前蒐集的邏輯限制式利用 solver 求解 (如：SMT[8]、Z3[9] 等等)，以取得欲執行該路徑所需的實際輸入值；透過這個流程理論上我們可以追蹤所有的執行路徑，探索是否有不當的輸入值能觸發程式崩潰。

如下 Figure 2 為一個簡單的 binary-based symbolic execution 執行時的狀況。它會依序讀入並根據指令進行相對應的模擬，位址 4004f1 將一個標記為 symbolic variable 的位址載入到暫存器 eax 中，則 symbolic execution engine 會在 constraint table 中標記 eax 為 sym_var1；4004f8 將 eax 加 3，symbolic execution engine 則會在 constraint table 中記錄 eax 加 3。當遇到分支跳轉指令時，如果無法決定該執行哪個分支，如 Figure 2 中，eax 的數值是一個 symbolic variable，當使用者輸入不同數值，將導致不

同的分支被執行，因此 symbolic execution engine 需要分別追蹤該分支為真和假時的狀態；由於每執行一條指令就返回顯得十分沒有效率，通常會讓 symbolic execution engine 遇到跳轉指令時，處理完狀態複製再返回，這樣的流程我們將其稱為一個步驟 (step)。

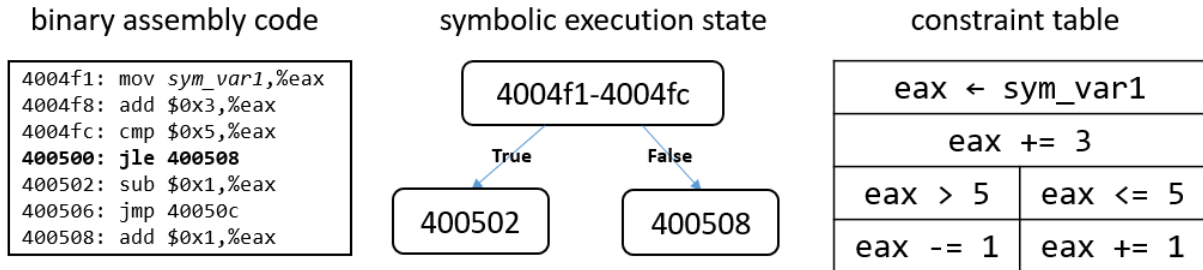


Figure 2: symbolic execution 流程

雖然 symbolic execution 理論上能夠探索所有執行路徑，但在探索的過程中會遭遇到上述的問題：當分支條件取決於 symbolic variable 時，很有可能兩邊的限制式都是可滿足的，這意味著程式必須複製並分別維護兩條路徑，直到發現路徑無解為止，當越來越多路徑需要被追蹤，就產生了路徑爆炸問題 (path explosion problem)。

2.2 Monte Carlo tree search

MCTS 是一種啟發式搜尋演算法，近年來最廣為人知的應用是遊戲 AI 方面的演算法；Monte Carlo 模擬是利用模擬和統計，得到一個近似解，在足夠大量的模擬下，理論上我們可以得到一個跟最佳解非常接近的答案。MCTS 套用了這種模擬的方式，維護一棵樹 (在遊戲 AI 中通常是一個遊戲盤面的狀態樹) 並統計每個盤面的勝率，期望能只探索部分的樹，而非全部探索完的情況下，就能知道該盤面的勝率。

在 [10] 中說明了 MCTS 演算法的基本流程，如 Figure 3：

- **Selection** 根據設定的 *Tree Policy*，從根節點開始遞迴性的決定一個目前最需要展開的節點，如 Figure 3 中的 Selection 部分，以粗框標記出的節點。可展開的節點在這裡定義為狀態尚未終止且有尚未訪問的子節點。
- **Expansion** 在選擇的節點上，執行一個合法的動作來新增子節點，如 Figure 3 中的 Expansion 部分，在挑選的節點下新增一個節點。
- **Simulation** 從這個新增的節點上使用 *Default Policy* 來進行模擬執行，產生結果。

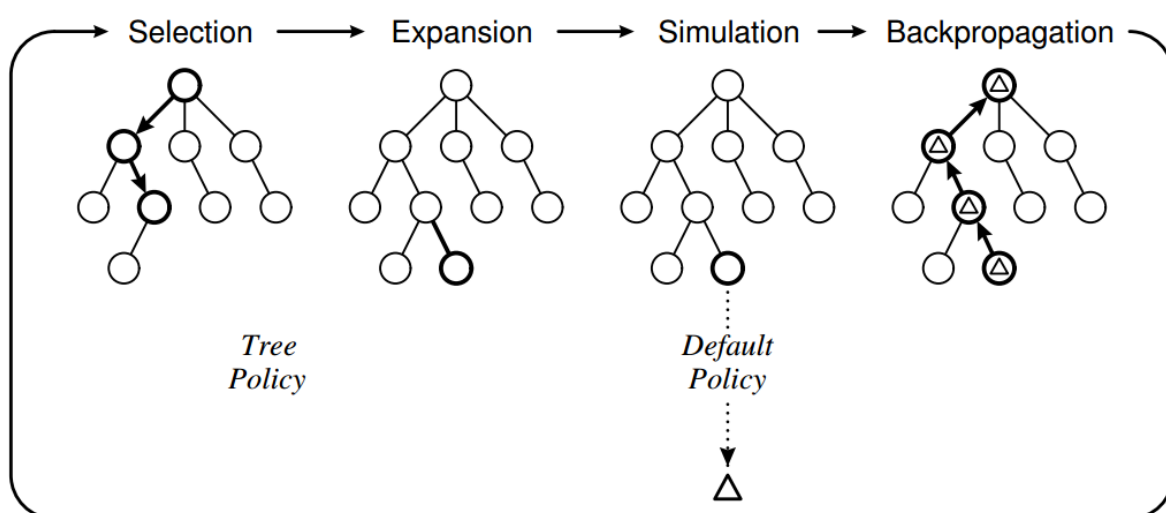


Figure 3: Monte Carlo Tree Search

- **Backpropagation** 模擬的結果會回饋到步驟 1 所選擇路過的那些節點上，更新他們的統計數據。

在這邊有兩個 policy：*Tree Policy* 代表的是如何決定要選擇和新增節點的演算法。*Default Policy* 則是如何從該節點的狀態模擬對局直到獲得勝負結果。雖然這兩個 Policy 也可以簡單的使用隨機方式決定，但適當的演算法有助於強化 MCTS 的準確度，如 [11] 便指出，*Tree Policy* 可以使用 upper confidence bound (UCB) 演算法來取代隨機挑選，如果我們把盤面的位置當成吃角子老虎機，視為 multi-armed bandit problem 來處理的話，會比原本的隨機選擇好；另外他也指出模擬時除了用這些簡單的方法，也可以使用更耗費資源的啟發式邏輯和評價方式，在對於 higher branching factor 的遊戲會有較好的效果。

Chapter 3

Method

3.1 Motivation example

在不同策略下，樹的走訪順序如 Figure 4所示，假設固定先選擇的都是左子樹，DFS 會優先選擇最深的節點來走訪，而 BFS 選擇最淺的節點，這兩個策略都有一樣的問題：順序是固定的，如 DFS 很可能因為左子樹太過龐大，但較高效益的節點在右子樹，而浪費很多時間在走訪較低效益的節點；又如 BFS，很可能較高效益的節點要往深處搜尋，但走訪深度較淺的節點就已經花費不少時間。它們共同的問題就是走訪到節點 5、6 的順序都一樣是在最後面。而 MCTS 的走訪順序是不固定的，它會動態的選擇一個被認為是目前最應該走訪的節點來進行計算。

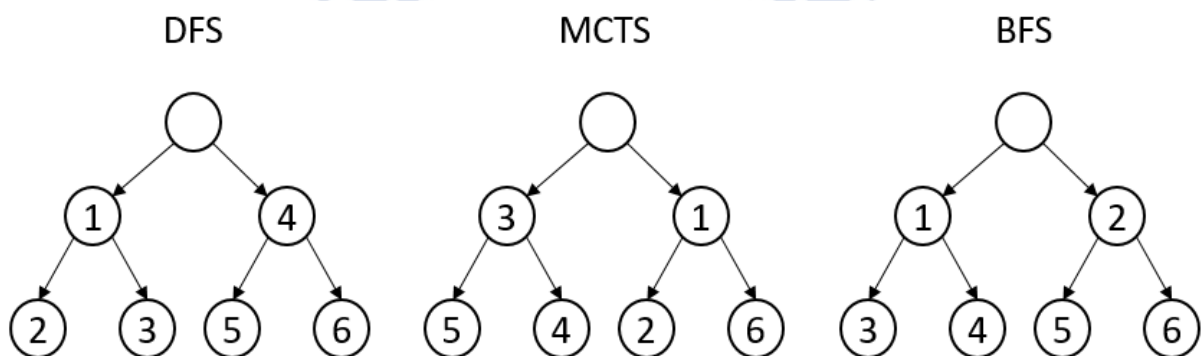


Figure 4: 不同策略的樹走訪順序

Figure 5為範例程式碼 Algorithm 1 在 symbolic execution 計算過程中所建立的狀態樹，每個節點都代表著一個狀態，包含了程式的暫存器和記憶體內容，symbolic variables 的 constraint 等等。root 節點代表剛進入該段程式的狀態，而其左子節點則是執行完第 1 行時的狀態，我們以數字 1 來標記該節點目前執行的位址，沒有標記數

字的節點則是代表程式結束，其他節點也以此類推。

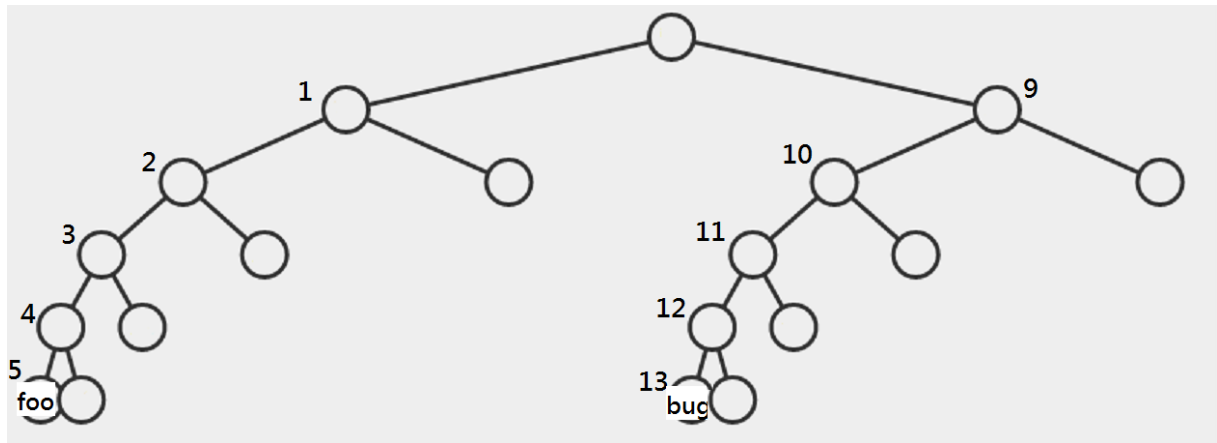


Figure 5: Tree of the example code

從 Figure 5 這棵樹中我們可以發現，當使用 DFS 來進行搜索時，程式需要先走訪完整個左子樹才有可能執行到 bug 函式，更壞的情況是有可能左子樹大到無法走完，程式被困在其中進行無效的運算；而如果使用 BFS 來搜索時，雖然最後 foo 函式和 bug 函式會在差不多的時間點被執行到，但很有可能因為在過程中已經發現太多節點，而造成記憶體空間不足的問題；所以我們希望採用一個策略，是程式能動態決定該選擇哪個節點來往下走訪的。

Algorithm 1 Example Code

```
1: if  $w > 5$  then
2:   if  $x > 10$  then
3:     if  $y > 20$  then
4:       if  $z > 30$  then
5:         foo()
6:       end if
7:     end if
8:   end if
9: else
10:  if  $x > 10$  then
11:    if  $y > 20$  then
12:      if  $z > 30$  then
13:        bug()
14:      end if
15:    end if
16:  end if
17: end if
```

對 symbolic execution 而言，不管採用哪種搜尋策略，我們的目標都是：如何達到更高的程式碼覆蓋率，以盡可能的測試出不同執行狀態時，程式是否會發生錯誤或崩潰。我們希望採用 MCTS 演算法來搜索時，能比傳統的搜尋方法能在同樣的資源限制下 (如時間限制、記憶體限制) 走訪更多的程式碼。

3.2 proposed algorithm

Algorithm 2 applying MCTS algorithm to symbolic execution

```
1: function SEARCH( $p_r$ )
2:   set  $p_r$  as root of Tree  $T$ 
3:    $B \leftarrow \emptyset$ 
4:   while within computational budget do
5:      $p \leftarrow \text{TreePolicy}(T)$ 
6:      $B \leftarrow B \cup p$ 
7:      $S \leftarrow \text{step}(p)$ 
8:     for each path  $p_c \in S$  do
9:        $V \leftarrow \text{DefaultPolicy}(p_c)$ 
10:       $Q(p_c) \leftarrow \alpha \frac{|V-B|}{N} + \beta |p_c|$ 
11:      add a new child  $p_c$  to  $p$ 
12:    end for
13:    BackPropagation( $p$ )
14:  end while
15: end function
```

Algorithm 2 為我們設計的演算法主體。 p_r 為現在要搜尋的路徑，首先將 p_r 設為

樹 T 的 root，以記錄路徑間的關係，並以集合 B 來計算路徑走訪的數量。在第 5 行我們會先利用 *Tree Policy* 從樹 T 中挑選一個應該計算的路徑 p ，並將 p 加進集合 B 中 (路徑 p 事實上可以被視為是數個走訪過的位址集合)，接著將 p 遞交給 symbolic execution engine 進行計算，symbolic execution engine 會將這條路徑 step (往下執行若干條指令，直到遇見分支跳轉指令才停止)；step 後可能產生數條路徑，我們以集合 S 來表示， S 中的路徑代表 p step 後可能的狀態，如執行 if 時或執行 else 時的狀態，遇到 switch case 時執行不同 case 的狀態；在第 8 行對 S 中的每條路徑 p_c 我們都會利用 *Default Policy* 來模擬其未來可能的走向，並評估此路徑的價值 $Q(p_c)$ ，將 p_c 標記為 p 的 child；最後在第 13 行整理先前第 8-12 行的資訊並記錄起來。

在第 10 行為我們計算路徑價值的公式，此公式分為兩個部分，其中 α 、 β 和 N 為可以人為控制的參數。 $\frac{|V-B|}{N}$ 中的 V 為路徑 p_c 經由 *Default Policy* 計算後得出未來可能會執行的位址集合，和集合 B 取差集後計算其數量，除以 N (*Default Policy* 模擬的次數) 就是路徑 p_c 增加程式執行區塊覆蓋率的期望值；而 $|p_c|$ 為 p_c 執行過的程式碼區塊數量，這個參數是為了在所有路徑的覆蓋率期望值都非常低的時候，能優先選擇較深的路徑避免進行無謂的探索。 α 和 β 是這兩個數值的權重。計算 Q 時的 α 主要控制的是增加覆蓋率的期望值，由於路徑的模擬是根據 Control flow graph (CFG) 來猜測，如果產生的 CFG 不正確 (不同的實作方式可能產生不同結果) 或程式的實際執行狀況和模擬的結果有落差，期望值就會變得不準確，相對的對於簡單的小程式，模擬的準確度有可能是較高的，因此適當的調整 α 可以修正模擬的數據；而 β 是為了因應遇到大量迴圈或 strcmp 這類函式的措施，由於進入迴圈容易產生大量的分枝，會讓 path 數量一下子成長很多，*Tree Policy* 在選擇時也容易被混淆，我們透過計算該 path 已經執行過的程式碼區塊數量，讓演算法在挑選時偏好已經執行比較多區塊數量的路徑。

Algorithm 3 是 Algorithm 2 中所提及的函式實作部分。*Tree Policy* 用來挑選應該被計算的路徑，而價值計算事實上是由 *BestChild* 決定，其中的一個參數 C ，其影響的是 MCTS 中最大的特徵：exploitation 和 exploration，也就是程式該往較深的點進行計算，還是選擇較少被計算過的點，不過這個數值在 branching factor 高時較有效，而我們產生出的 path 常常只有 1 至 2 個，所以這個參數的影響並不大，唯一會影響的是當某個節點尚未被計算過任何一次的時候， $N(p_c)$ 為 0，根號內的數值會是無限大，程式就一定會選擇該節點來進行計算。*Default Policy* 的參數 N, M ， N 是要進行幾次模擬，而 M 是避免程式進入無窮迴圈，當到達一定數字時會強制中斷模擬，對於較小的程式可以選擇較小的數字，而較複雜的程式可以選擇比較大的數字來增加其準確性，但也相對的花費更多時間。*BackPropagation* 會更新兩項數值， $N(v)$ 為 v 被訪問的次

數，對於價值 $Q(v)$ 則以 v 的子節點們的平均來替代。

Algorithm 3 Policies for our algorithm

```

function TREEPOLICY( $T$ )
   $n \leftarrow$  root of  $T$ 
  while  $n$  is not terminated do
    if  $n$  is expandable then
      return  $n$ 
    else
       $n \leftarrow \text{BestChild}(n, C)$ 
    end if
  end while
end function

function DEFAULTPOLICY( $p$ )
   $V \leftarrow \emptyset$ 
  for  $i = 1; i < N; i++$  do
     $v \leftarrow$  find vertex at CFG( $p$ 's addr)
    for  $j = 1; (j < M) \text{ and } (v \text{ has any edge}); j++$  do
      add  $v$  to  $V$ 
       $v \leftarrow$  random pick a vertex which  $v$  directed to
    end for
  end for
  return  $V$ 
end function

function BESTCHILD( $p, C$ )
   $Q_{max} \leftarrow \arg \max_{p_c \in p} Q(p_c)$ 
  return  $\arg \max_{p_c \in p} \frac{Q(p_c)}{Q_{max}} + C \sqrt{\frac{2 \ln N(p)}{N(p_c)}}$ 
end function

function BACKPROPAGATION( $v$ )
  while  $v$  is not null do
     $N(v) += 1$ 
     $Q(v) \leftarrow$  average of  $Q(v$ 's children)
     $v \leftarrow$  parent of  $v$ 
  end while
end function

```

3.3 algorithm explanation

如下 Figure 6 為 MCTS 簡要的執行過程示意圖。在 Step 1 設定根節點後，由於沒有任何子節點，我們在 Selection 階段便直接選擇根節點，進行 Expansion 後新增兩個

子節點，其價值經 Simulation 計算後分別為 60 和 80。Step 2 從根節點找到兩個子節點，在這邊選擇價值為 80 的子節點進行 Expansion，得到兩個子節點並計算出其價值為 50 和 52，獲得子節點的估計價值後，便可以作為 Backpropagation 使用，節點 1 的價值將改由其子節點價值的平均來取代，如這邊即從 80 變為 51。Step 3 因為在 Step 2 時修正了節點 1 的價值，在 Selection 階段便會轉而選擇節點 2，經過 Expansion 和 Simulation 後同樣利用子節點來修正節點 2 的價值，不斷的重複這個步驟來挑選最應該被計算的節點。

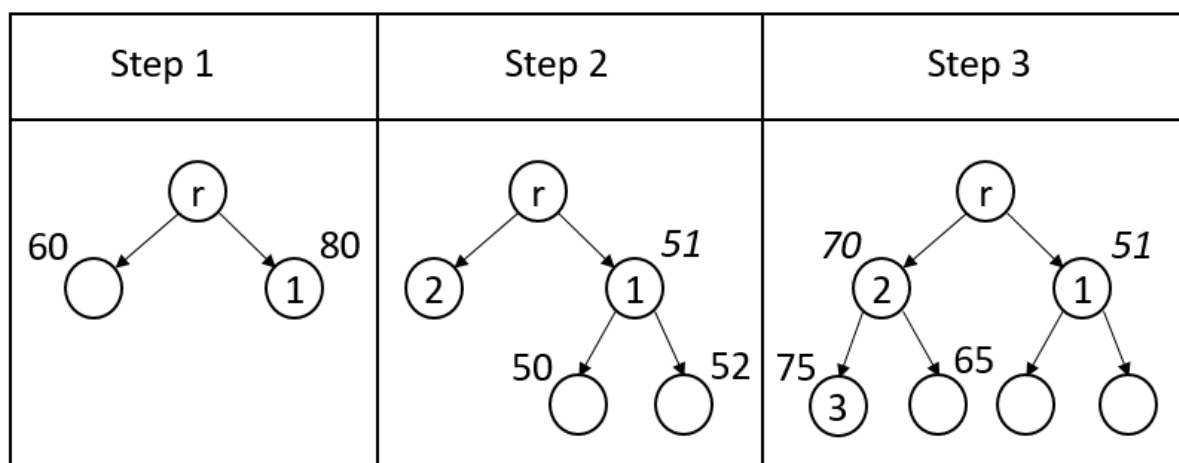


Figure 6: MCTS 執行過程

Chapter 4

Implementation

4.1 Angr's architecture

Angr[12] 是一個支援多種架構的程式分析平台，主要以 python 撰寫，能夠執行如 dynamic symbolic execution 和多種靜態分析方法。這個平台可以自動替使用者完成一些工作，如：讀取並載入程式 binary 到分析平台中、將程式 binary 轉換成 intermediate representation (IR)，程式靜態分析，如：相依性分析、control flow graph、data flow graph。Angr 主要擁有以下幾個元件：binary loader 可以載入不同平台的程式、Intermediate representation 讓不同格式的程式都能使用同一套演算法或工具、Solver Engine 以計算出 symbolic execution 中的變數值、Program states 以模擬機器的狀態，如暫存器目前的值、Program paths 以記錄程式執行的路徑作為分析用、symbolic execution 和 program analysis。

要使用 Angr 進行程式分析必須自行撰寫 python script 來呼叫相關功能，如 Figure 7 為使用 Angr 執行 symbolic execution 的範例 script；與作業系統執行 process 的過程類似，首先執行 Line 22，必須找到程式的 binary 位置並傳給 Angr 進行初始化，由於程式大多會連結多個函式庫，如果將其視為程式的一部份一同載入會使得載入時間變得很長，因此在 Line 14 選擇關閉一同載入的功能，直到程式執行到該地方了再去載入函式庫；接著必須使用 Angr 內的 solver engine：claripy 來設定 symbolic variables，以程式 gif2png 為例，執行的時候只需要給他兩個命令列選項，而第二個選項為圖片位置，所以我們將第一個選項設為 symbolic variable，使用 claripy 的 BVS (symbolic bitvector) 物件，並給它一個名稱和初始值；再來的 Line 16 則是將設定好的程式啟動並執行到某個狀態，當程式要執行時除了讀取程式 binary 外還要設定它的記憶體區段、初始化數值、環境變數等等資訊，而我們所寫的 entry 是指這些資訊

都設定完成時，程式剛要從程式進入點開始執行的狀態；另外我們還可以給它一些其它設定，如命令列參數傳入了剛剛宣告的 symbolic variable，還有檔案的名稱，也可以設定執行時要開啟或關閉哪些選項，在這邊關閉了 LAZY_SOLVES 功能，讓 Angr 會自動移除 unsatisfiable 的路徑；這些前置工作都完成後才獲得預備要進行 symbolic execution 的初始化狀態，接著在 Line 18 使用 Angr 的 path_group 功能來代替我們管理 symbolic execution 的繁瑣設定；path_group 代表的是一群 path 的集合，當 symbolic execution 在執行時會不斷的增長出新的 path，而 path_group 可以將 path 進行分類，如：errored、unconstrained、pruned、deadended、unsatisfiable 等等，此外還能過濾、合併、複製路徑，或是設定 symbolic execution 要停止的條件，是一個功能相當強大的 class。最後利用 path_group 的 run function 就可以幫我們執行 symbolic execution。



```
1  #!/usr/bin/python
2
3  import sys
4  import angr
5  import simuvex
6  import logging
7
8  # gif2png [-bdfghinprsvw0] [file.[gif]...]
9
10 logging.getLogger("angr.path_group").setLevel("DEBUG")
11 logging.getLogger("angr.DFS").setLevel("DEBUG")
12
13 def main(name, method, limit):
14     proj = angr.Project(name, load_options={"auto_load_libs": False})
15     argv1 = angr.claripy.BVS("argv1", 0xE * 30)
16     initial_state = proj.factory.entry_state(args=[name, argv1, "fake.gif"], remove_options={simuvex.s_options.LAZY_SOLVES})
17
18     path_group = proj.factory.path_group(initial_state, method=method, limit=limit)
19     path_group.run()
20
21 if __name__ == '__main__':
22     main('/usr/bin/gif2png', sys.argv[1], int(sys.argv[2]))
```

Figure 7: Example script to run symbolic execution in Angr

4.2 Exploration techniques

前一節提到了 path_group，在它內部也實作了一套方法可以讓使用者撰寫不同的 exploration technique 來應用到 symbolic execution 上，在提到 exploration technique 前必須先提及的是 path_group 內執行 symbolic execution 的方法，我們呼叫的方法是 run，其內部呼叫了 step 使其進行無數次迴圈，如 Figure 8 為部分原始碼，run 會將 n 設為一個極大值；而 step 真正呼叫的是 __one_step function，執行後檢查是否還需要

繼續執行，不斷進行迴圈。

```
for i in range(n):
    l.debug("Round %d: stepping %s", i, pg)

    pg = pg._one_step(stash=stash, selector_func=select_func)
    if step_func is not None:
        pg = step_func(pg)

    if len(pg.stashes[stash]) == 0:
        l.debug("Out of paths in stash %s", stash)
        break

    if until is not None and until(pg):
        l.debug("Until function returned true")
        break
```

Figure 8: partial code of step in Angr

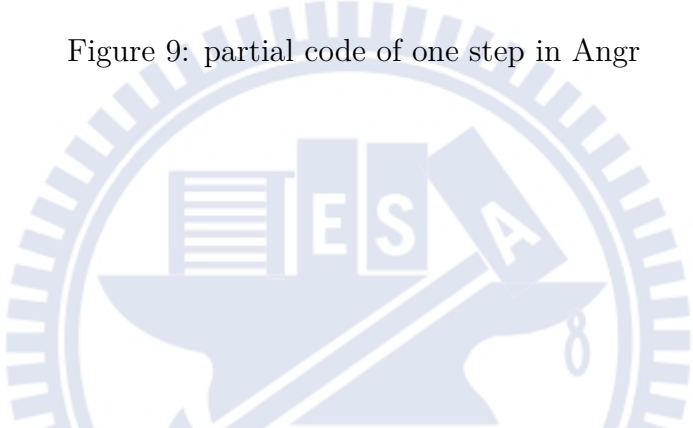
`_one_step` function 內才是真正會呼叫 exploration techniques 提供的方法之處，在啟用每個 exploration technique 時，Angr 會將它們提供的四種方法：`step`、`step_path`、`filter`、`complete` 給加入 hook function 內，如 Figure 9 中的變數 `_hook_step` 為一個陣列，裡面存放不同 exploration techniques 提供的 step function，首先它會將一個 step function 取出，並執行該 step function，而 exploration technique 提供的 step function 需要再呼叫一次 step，這時如果還有其他 hook 的 step 就會取出來繼續執行，否則就會執行原生的 step function，如 Figure 10 為 Exploration technique DFS 的 step function 實作，Line 16 先呼叫 step 後，才執行其對路徑的重新選擇。我們可以將 exploration technique 的 step 分為兩個部分，一為 step 前處理，一為 step 後處理，論文中的 MCTS 演算法實作完全是環繞於這兩個區塊而成。

```

if len(self._hooks_step) != 0:
    # hooking step is a bit of an ordeal, how are you supposed to compose stepping operations?
    # the answer is that you nest them - any stepping hook must eventually call step itself,
    # at which point it calls the next hook, and so on, until we fall through to the
    # basic stepping operation.
    hook = self._hooks_step.pop()
    pg = self.copy() if self._immutable else self
    pg._immutable = False # this is a performance consideration
    out = hook(pg, stash, selector_func=selector_func, successor_func=successor_func, check_func=check_func)
    out._immutable = self._immutable
    self._hooks_step.append(hook)
    if out is not self:
        out._hooks_step.append(hook)
    return out

```

Figure 9: partial code of one step in Angr



```

14 def step(self, pg, stash, **kwargs):
15
16     pg = pg.step(stash=stash, **kwargs)
17     if len(pg.stashes[stash]) > 1:
18         pg.stashes['deferred'].extend(pg.stashes[stash][1:])
19         del pg.stashes[stash][1:]
20
21     if len(pg.stashes[stash]) == 0:
22         if len(pg.stashes['deferred']) == 0:
23             return pg
24         i, deepest = max(enumerate(pg.stashes['deferred']), key=lambda l: len(l[1].trace))
25         pg.stashes['deferred'].pop(i)
26         pg.stashes[stash].append(deepest)
27
28     return pg

```

Figure 10: partial code of exploration technique(DFS) in Angr

4.3 Applied MCTS into Angr

Figure 11是 MCTS 在 step 前所作的處理，對於接下來要用 symbolic execution 計算的路徑，會先取出該路徑的歷史路徑位址集合，並更新到目前總共執行過的位址集合中，這個集合在之後會用來判斷路徑是不是有可能增加覆蓋率，由於每次都只會有一條路徑在 stash 中準備要進行計算，所以我們只取出一條即可。

```
53     def step(self, pg, stash, **kwargs):
54
55         # 已經走過的路徑 加進total_cover裡
56         self.count += len(pg.stashes[stash])
57         self.total_cover.update(self._get_past_hist(pg.stashes[stash][0]))
58
59         addr_before = pg.stashes[stash][0].addr
60         addr_bef_name = self.fm.function(addr=addr_before)
61
62         # expansion
63         pg = pg.step(stash=stash, **kwargs)
```

Figure 11: MCTS of pre step in Angr

呼叫完原本的 step function 後，就等同於演算法中的 Expansion 階段已經完成了，之後會進行 MCTS 的計算處理，如 Figure 12，現在 stash 中的路徑都是剛剛進去計算的路徑的子節點，我們會將每一條都取出來，並進行 Simulation，得到平均可增加覆蓋率後，將子節點增加到 tree 中。之後呼叫 tree 的 refresh_tree function，也就是演算法的 backpropagation 階段。最後才執行 Selection，雖然是為了配合 Angr 的設計，但在演算法上步驟的調整並不影響計算，在這個階段會呼叫 tree 的 select_node function 幫我們找出最好的那個節點。

```

77     # move all path to income
78     if len(pg.stashes[stash]) > 0 and self.method != "DEFAULT":
79         for a in pg.stashes[stash]:
80             a.info.clear()
81             pg.stashes['income'].extend(pg.stashes[stash][:])
82             del pg.stashes[stash][:]
83
84     for a in pg.stashes['income']:
85         if 'rate' not in a.info:
86             a.info['rate'] = []
87             a.info['cover'] = self._get_past_hist(a)
88             for times in range(50):
89                 hist = self.simulation(a)
90                 a.info['rate'].append(len(hist))
91                 avg_cover = sum(a.info['rate']) / float(len(a.info['rate'])) \
92                     if len(a.info['rate']) != 0 else len(a.info['cover'])
93                 if self.method == "MCTS":
94                     self.tree.add_child(data=a, coverage=self.a*avg_cover+self.b*len(a.trace))
95
96     if self.method == "MCTS":
97         self.tree.refresh_tree()
98
99     if self.method != "MCTS":
100         pg.stashes['deferred'].extend(pg.stashes['income'][:])
101     del pg.stashes['income'][:]
102
103     if len(pg.stashes[stash]) == 0:
104         # if len(pg.stashes['deferred']) == 0:
105         #     return pg
106         if self.method == "DFS":
107             i, deepest = max(enumerate(pg.stashes['deferred']), key=lambda l: len(l[1].trace))
108             pg.stashes['deferred'].pop(i)
109             pg.stashes[stash].append(deepest)
110         elif self.method == "BFS":
111             i, deepest = max(enumerate(pg.stashes['deferred']), key=lambda l: -len(l[1].trace))
112             pg.stashes['deferred'].pop(i)
113             pg.stashes[stash].append(deepest)
114         elif self.method == "MCTS":
115             node = self.tree.select_node()
116             pg.stashes[stash].append(node.data)
117         elif self.method == "DEFAULT":
118             pass
119         else:
120             1.error("Unsupport method %s", self.method)

```

Figure 12: MCTS of post step in Angr

Chapter 5

Result and Evaluation

5.1 Environment

我們將本篇論文提出的演算法實作於 Angr (一個開源的 python 符號執行框架) [12] 上，並在 Ubuntu 16.04 作業系統環境進行實驗，受測程式如 Table 1。硬體環境使用的是 Intel i7-2600k 處理器以及 24 GB 記憶體。主要比較的目標是演算法經由 symbolic execution 所執行過的程式碼區塊數量：程式碼區塊指的是一段程式碼中間沒有任何的跳轉指令，而以跳轉指令為結尾，也就是說程式碼區塊執行直到最後一個指令時才會跳轉到其他區塊，每個區塊都不會發生與其他區塊有所重疊或覆蓋的情形；我們以程式碼區塊的第一條指令位址為其開始位址並作為判斷是否重複的標準，對重複執行過的區塊我們只會計算一次而不是會重複計算多次。在前章所提及的演算法參數，由於參數調校並非本篇論文的重點，因此在所有實驗中都固定使用同一組參數來進行。

Table 1: Target Program's name and version

program name	version
cp	8.25
echo	8.25
hostname	3.16
ls	8.25
mkdir	8.25
ps	3.3.10
readelf	2.26.1
touch	8.25
cpp-markdown	1.00
gif2png	2.5.8

5.2 演算法與其他方法的比較

為了比較不同方法間的效率好壞，我們給定一個資源限制：程式最多只能使用 symbolic execution engine 來 step 3000 次，對時間和記憶體用量皆未限制。Table 2 是在此限制下 MCTS 和 DFS、BFS 比較的結果。其中數量為前一節所述的程式碼區塊數量，時間為該次實驗的執行時間 (秒)，效率則是區塊數量除以時間後得到每秒會增加的平均值。

Table 2: Comparison of the efficiency with different strategies

program	MCTS			DFS			BFS		
	#	time(s)	eft	#	time(s)	eft	#	time(s)	eft
cpp-markdown	566	272.07	2.08	529	306.27	1.73	478	318.16	1.50
cp	388	3732.45	0.10	218	2340.34	0.09	174	326.38	0.53
echo	84	164.90	0.51	76	283.49	0.27	83	126.61	0.66
gif2png	105	192.66	0.55	64	2115.42	0.03	59	680.98	0.09
hostname	175	277.80	0.63	62	147.08	0.42	61	309.11	0.20
ls	393	715.55	0.55	189	1053.82	0.18	152	3057.35	0.05
mkdir	320	275.96	1.16	82	142.86	0.57	119	198.10	0.60
ps	99	682.35	0.15	185	176.52	1.05	109	1083.84	0.10
readelf	288	870.36	0.33	393	913.68	0.43	156	1179.49	0.13
touch	302	331.78	0.91	213	1665.38	0.13	124	194.40	0.64
avg.	272	751.59	0.70	201.1	914.49	0.49	151.5	747.44	0.45
SD	148.51	1020.75	0.55	145.99	809.66	0.50	114.97	847.74	0.42

為了方便比較我們將數值繪製成折線圖，如 Figure 13，可以看到 MCTS 在大部分的實驗都是勝出的，只有兩隻程式的數值較差。接著比較其效率，結果如 Figure 14，除了原本的兩隻程式外，還有兩隻程式遜於 BFS，但仍贏過 DFS。雖然各有勝負，但從數據的平均值來看，MCTS 的效能是比較好的，而標準差也可以看出對實驗結果的不確定性，MCTS 相對於其他兩個方法並沒有極大的差異。



Figure 13: block coverage in small test

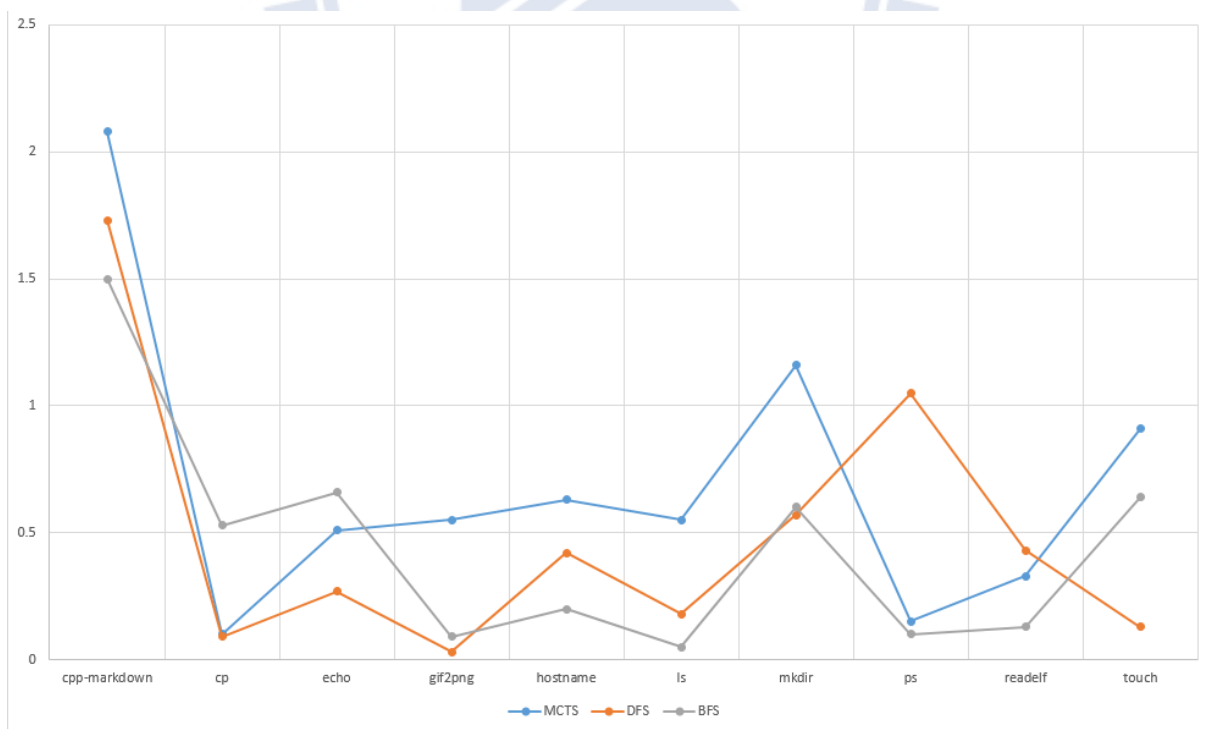


Figure 14: efficienct in small test

5.3 MCTS 的評估函式效能測量

在演算法中的 Selection 階段會遞迴性的選擇最好的子節點直到發現一個可展開的節點，而 Simulation 階段也會根據先前展開的節點來模擬評估其價值，這兩個階段在演算法中位居引導演算法的重要角色，因此我們想要知道如果沒有這兩個階段之下，演算法的效能如何，下面比較的對象是將 Selection 從選擇 BestChild 改為隨機選擇一個子節點，結果如 Table 3所示，結果只有程式 echo 和 ps 略輸於 Random，但差距非常的小，只有 3.5%，單從區塊涵蓋數量來看，平均有 35% 的成長，而 Random 甚至略遜於 DFS 的平均值，除上執行時間後，效率更加糟糕，這有可能是因為 symbolic execution 本身的優化技巧在隨機選擇後沒有辦法發揮效果，以至於平均執行時間增加到 2.7 倍。

Table 3: Comparison of the efficiency with random selection

program	MCTS			Random		
	#	time(s)	eft	#	time(s)	eft
cpp-markdown	566	272.07	2.08	479	2013.28	0.24
cp	388	3732.45	0.10	265	2038.27	0.13
echo	84	164.90	0.51	87	575.15	0.15
gif2png	105	192.66	0.55	101	1116.26	0.09
hostname	175	277.80	0.63	94	826.82	0.11
ls	393	715.55	0.55	168	3374.23	0.05
mkdir	320	275.96	1.16	187	1154.84	0.16
ps	99	682.35	0.15	103	1631	0.06
readelf	288	870.36	0.33	263	6599.76	0.03
touch	302	331.78	0.91	246	1025.58	0.24
avg.	272	751.59	0.70	199.3	2035.52	0.13
SD	148.51	1020.75	0.55	115.13	1703.61	0.07

5.4 長時間執行的效率比較

經小規模實驗得到 MCTS 較佳的結果後，我們用較大規模的方法來實驗，前項實驗的執行時間大約為 13 分鐘上下，且記憶體用量幾乎不會超過 2 GB；因此在這個實驗我們改以 24 小時的時間限制和 20 GB 的記憶體用量限制，而不限制其在 symbolic

execution engine 的 step 次數，來觀察在有限的記憶體用量下，我們的方法相較於傳統的方法能有多少幅度的效率成長。

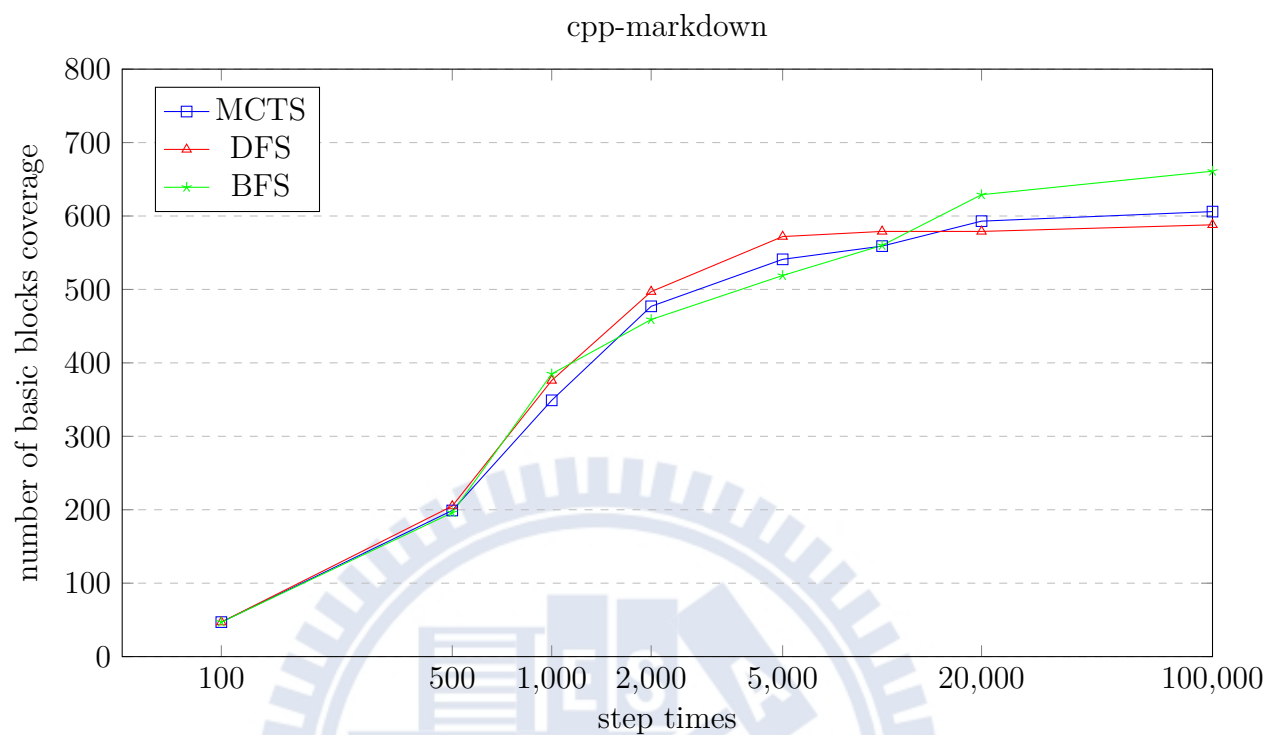


Figure 15: large test for cpp-markdown

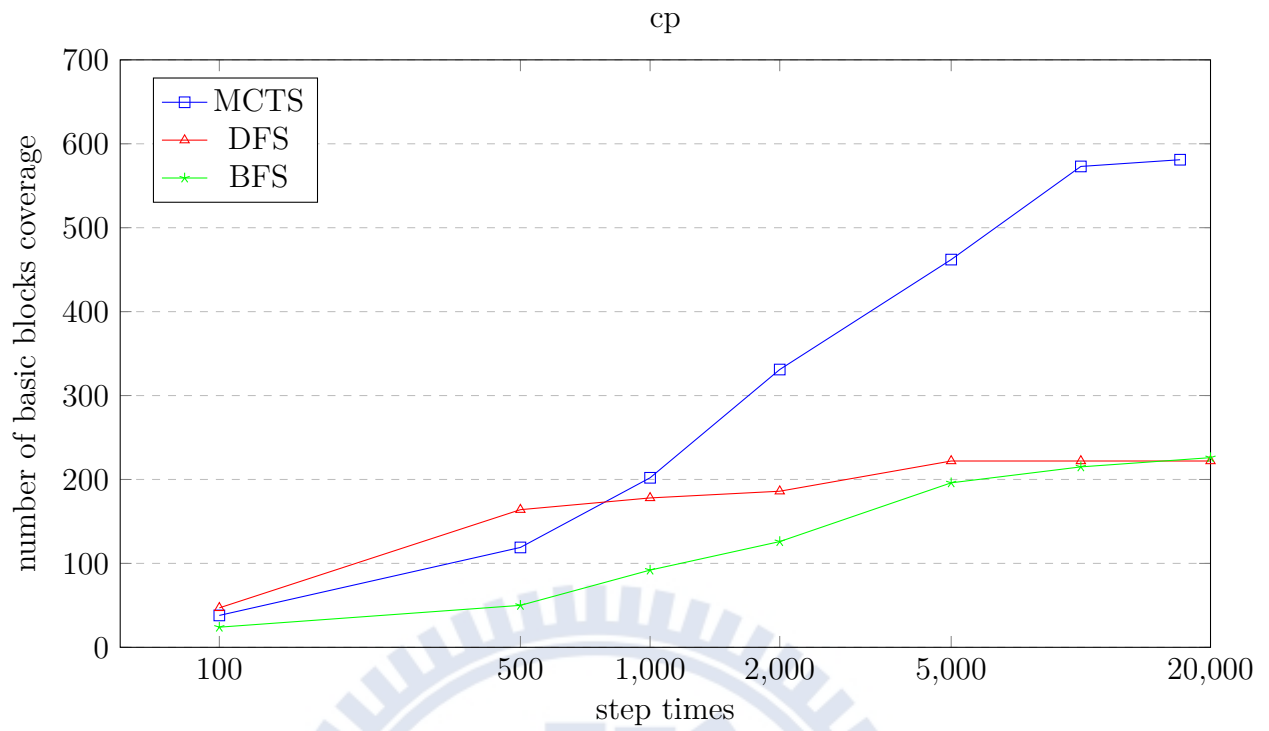


Figure 16: large test for cp

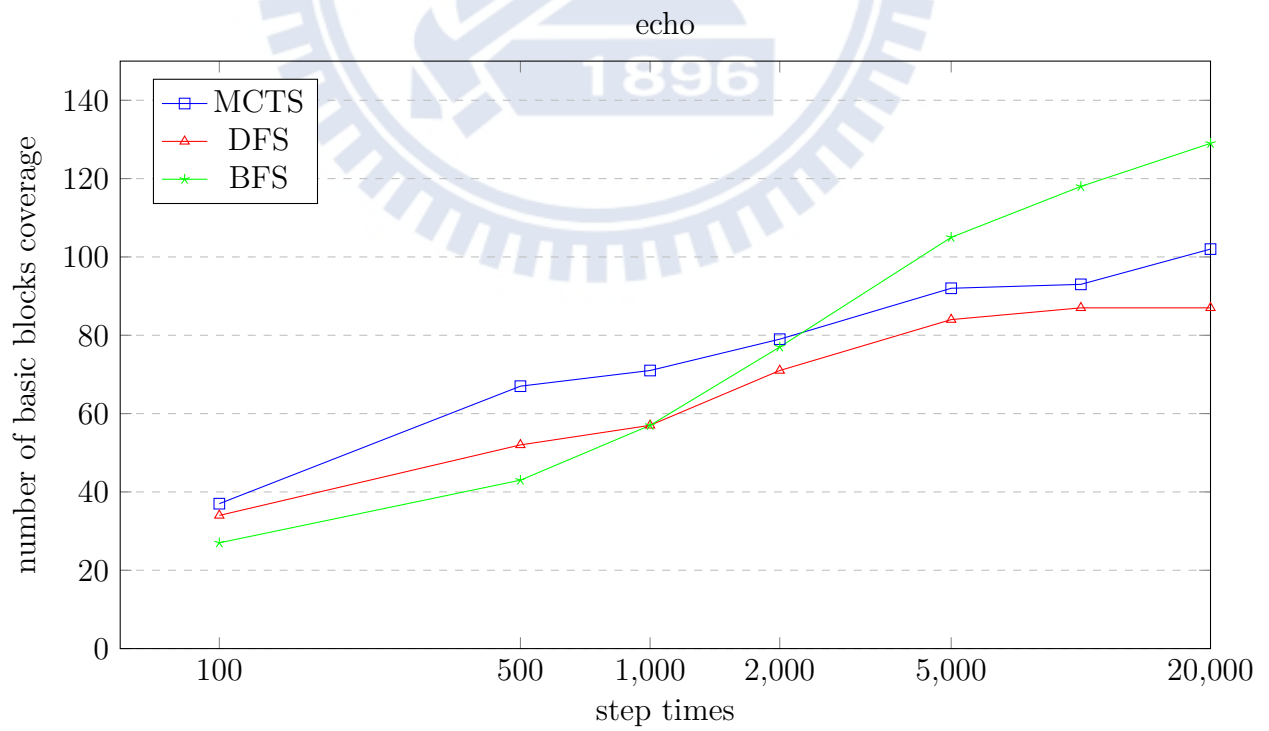


Figure 17: large test for echo

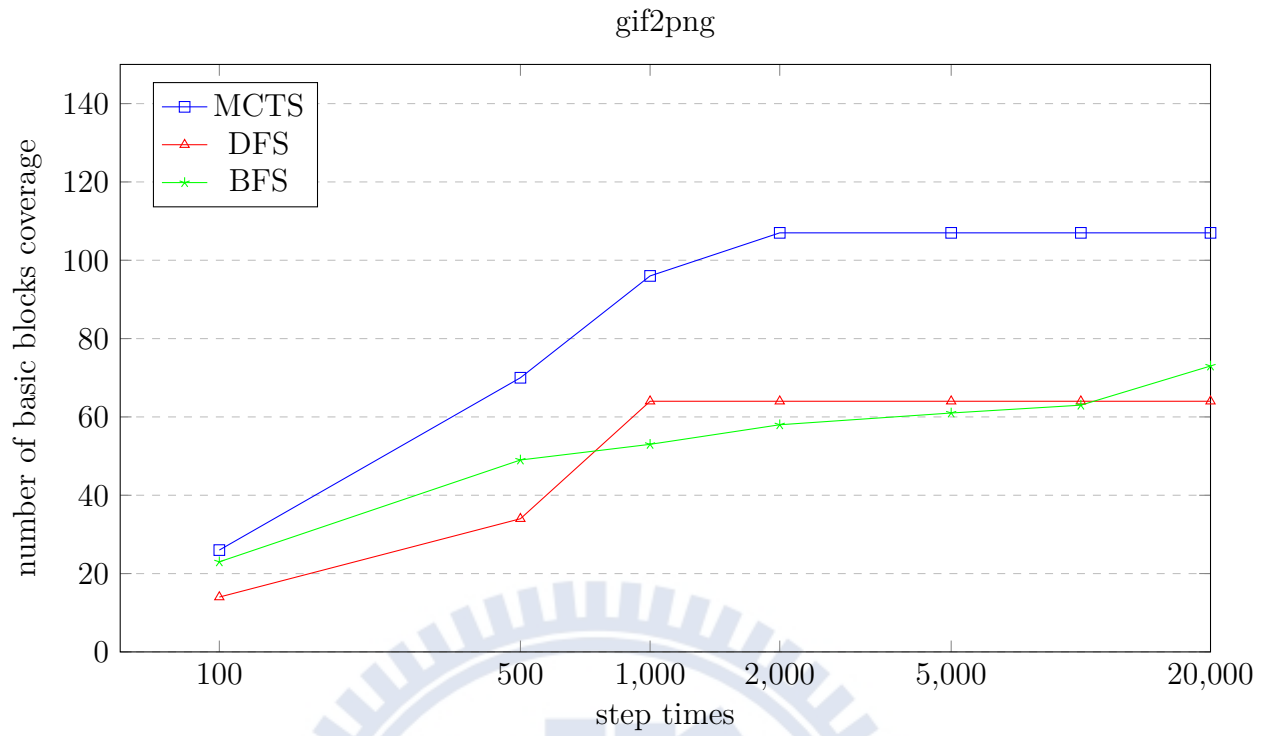


Figure 18: large test for gif2png

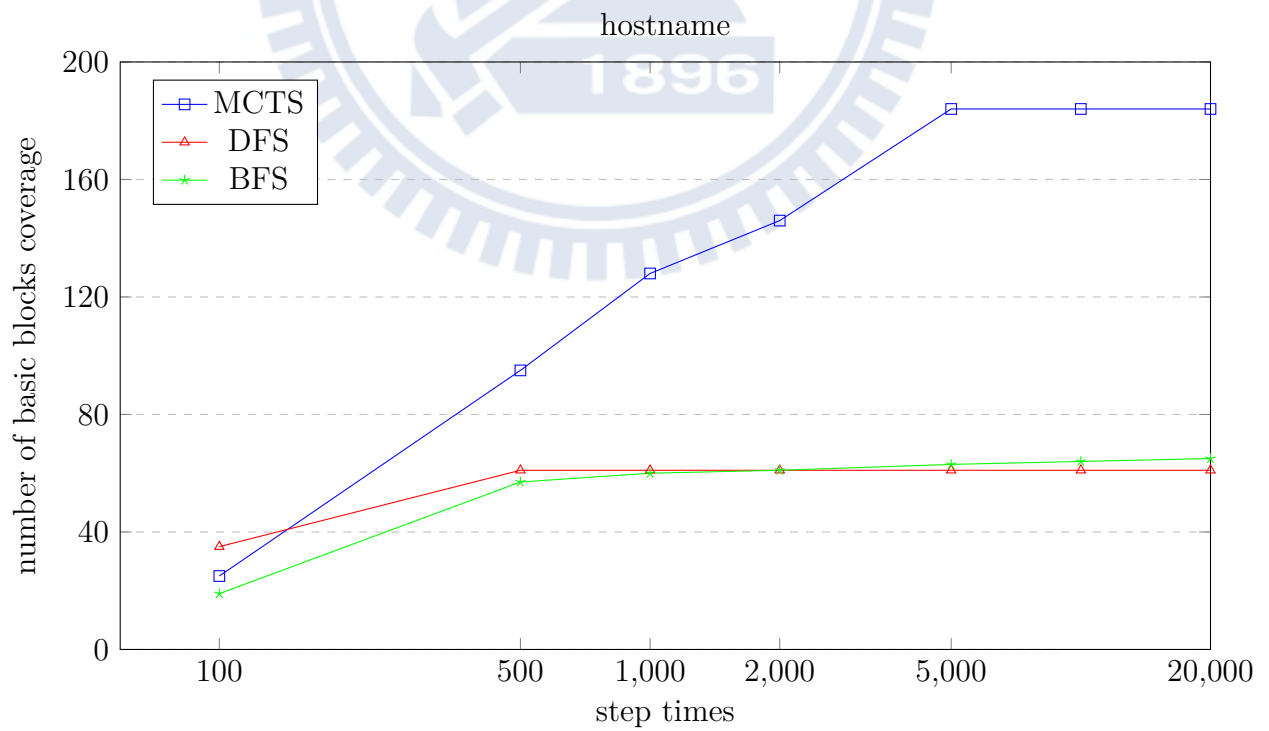
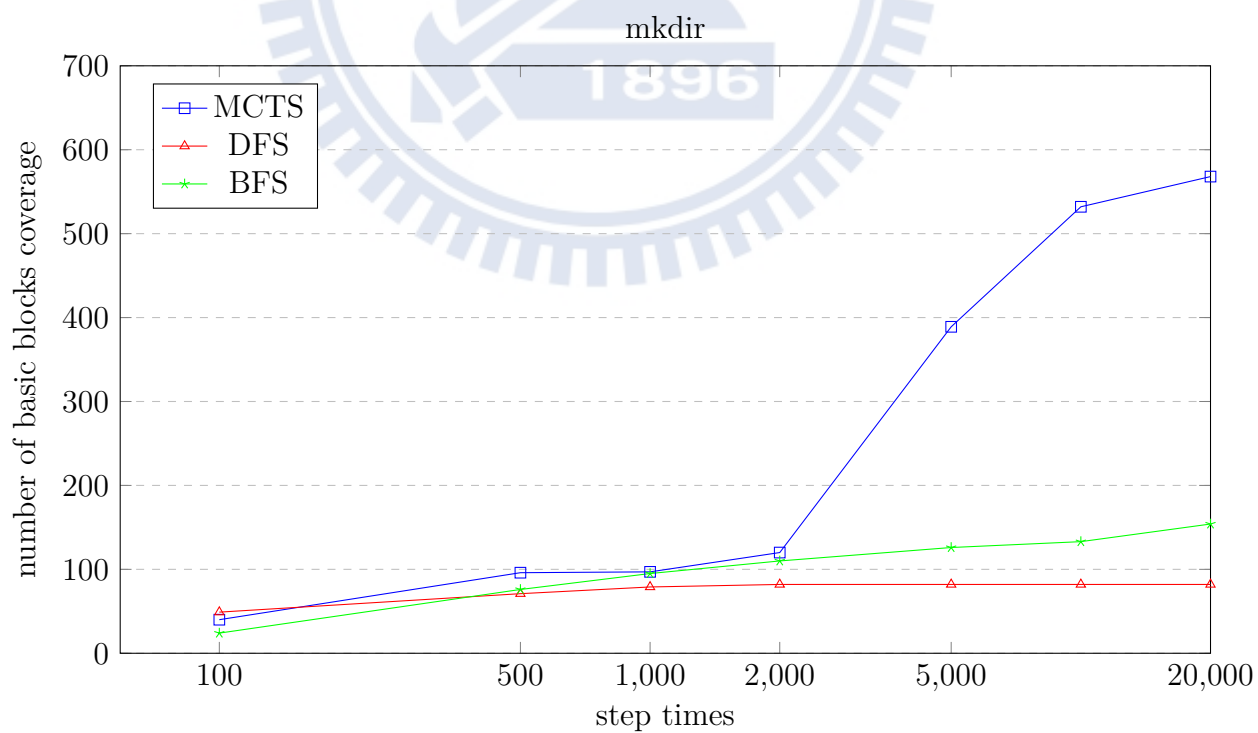
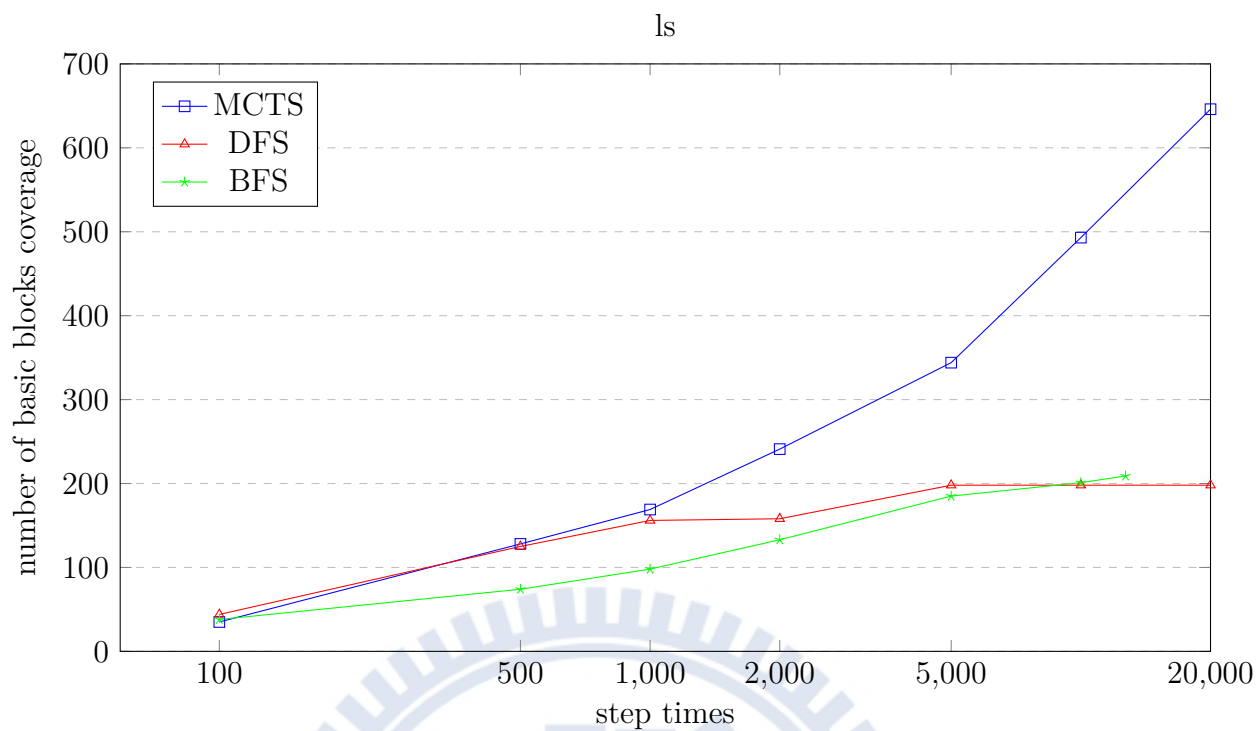


Figure 19: large test for hostname



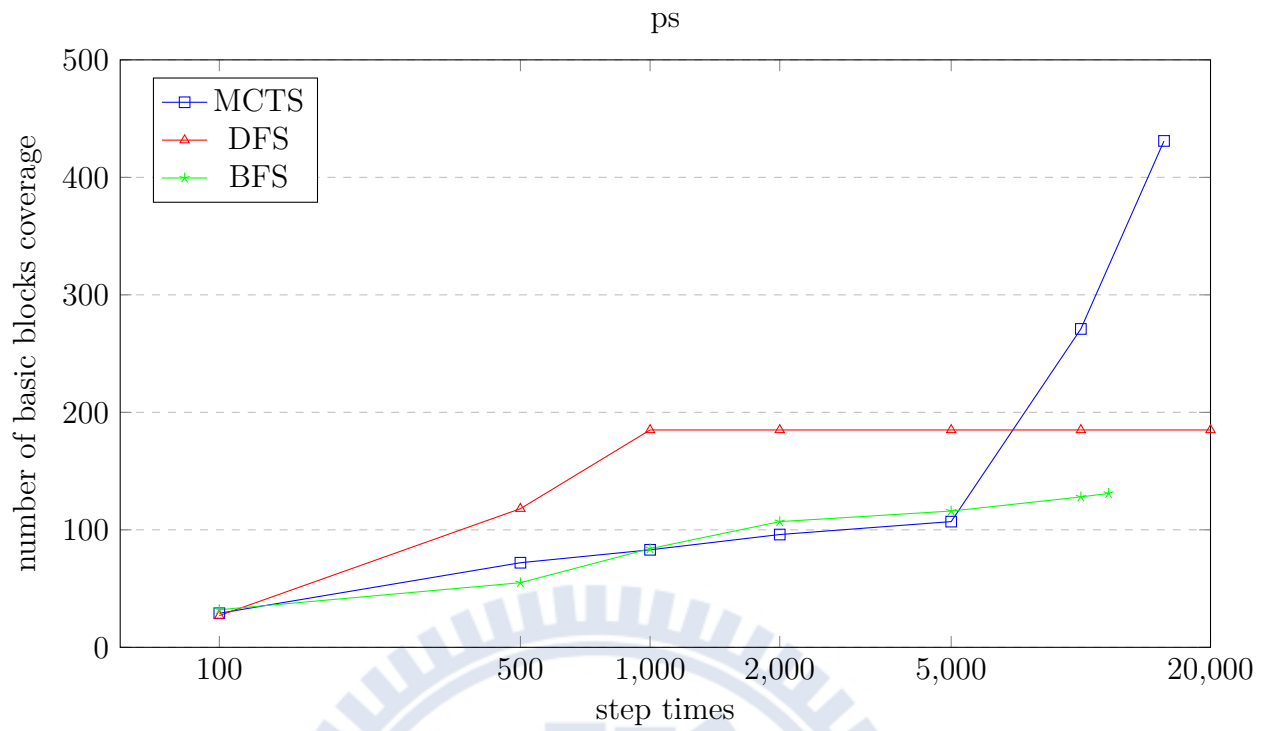


Figure 22: large test for ps

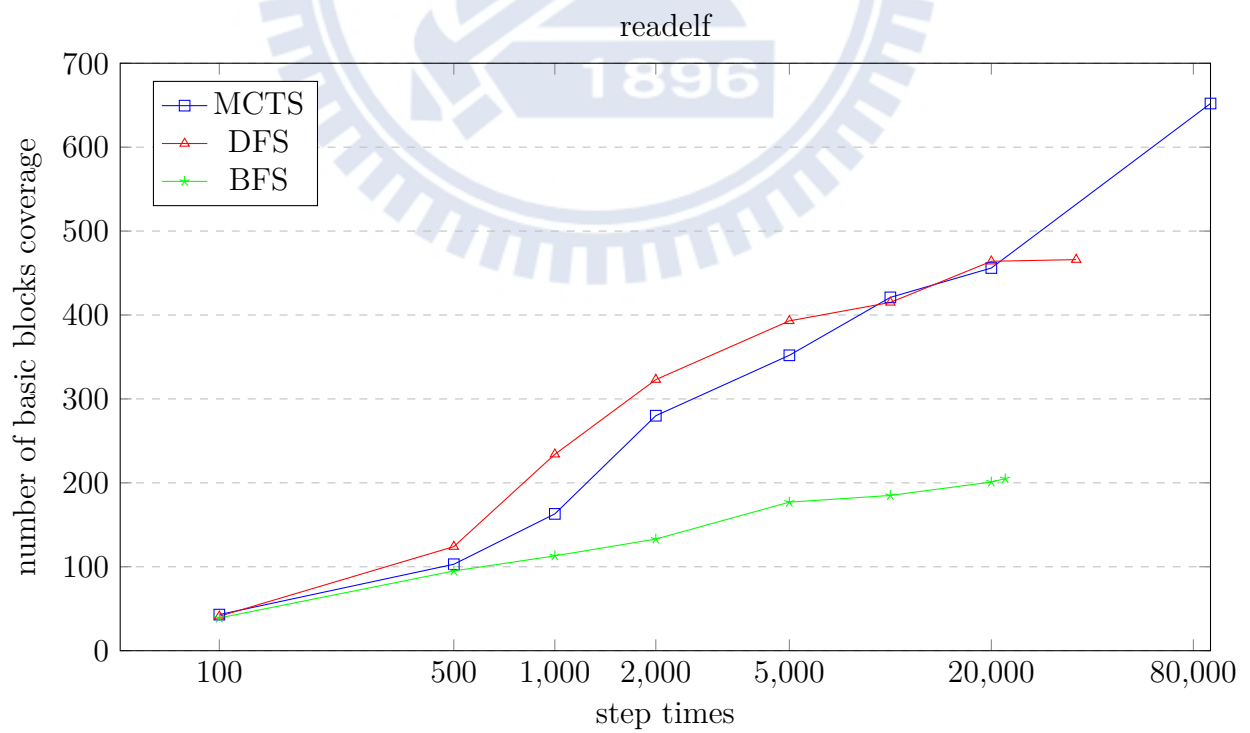


Figure 23: large test for readelf

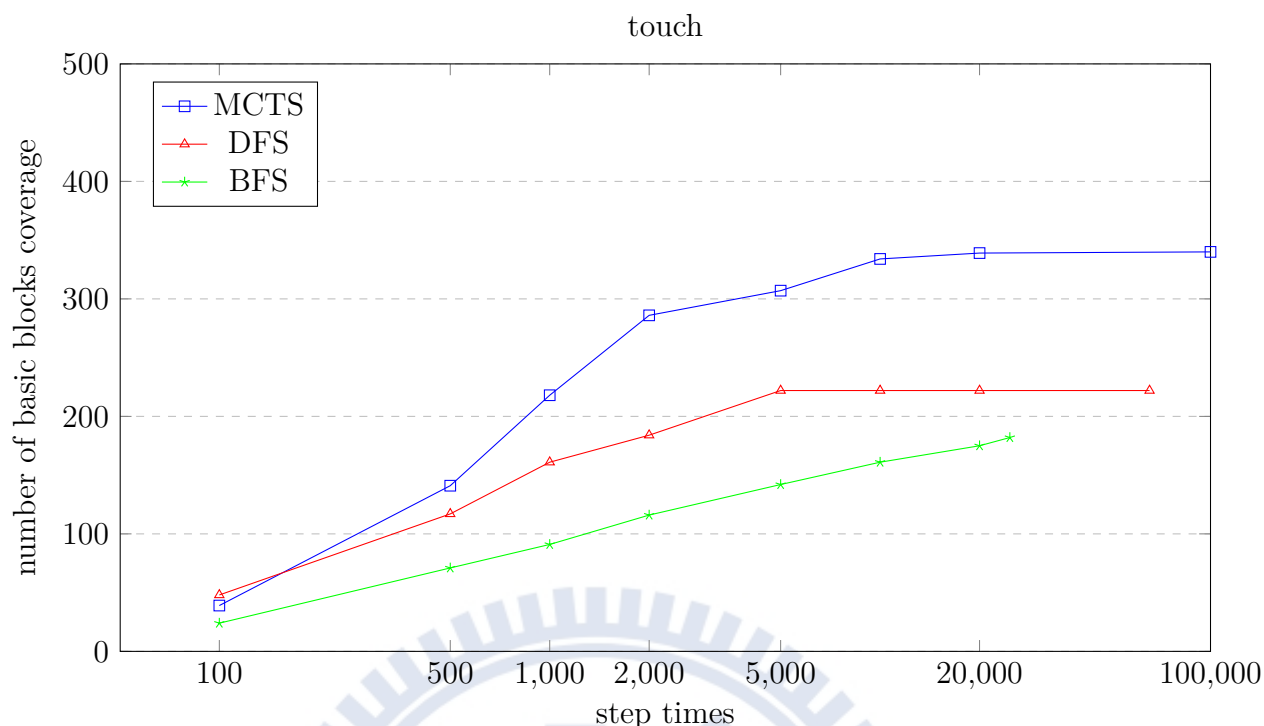


Figure 24: large test for touch

5.5 Discussions

從個別的實驗數據中可以看到 MCTS 在面對不同程式所發揮的功用。數值差異不大的例子有 Figure 15、17，成長情況不相上下，但後期 BFS 有著小幅領先。Figure 15 的線圖變化幾乎不相上下，而到最後 BFS 有較好的表現，可能的成因在於當 MCTS 和 DFS 同樣往較深的節點進行搜尋時，並沒有較顯著的效果，而 BFS 卻能涵蓋那些深度較淺的節點；而 Figure 17 相較之下，BFS 更明顯的有後來居上的表現，因此我們判斷很可能是另外兩種策略被困在特定迴圈內時，BFS 比較不容易被困在迴圈裡。

Figure 16、18、19、20、21 屬於 DFS 和 BFS 都會卡住不再成長，而 MCTS 仍可繼續成長的情況，這也是在所有程式裡面占最多的例子，表示 MCTS 充分發揮了它的特性「exploitation and exploration」，DFS 和 BFS 分別以一個固定的方式在選擇節點來運算時，MCTS 可以從中目前已有的節點中，挑選較有價值的路徑來進行運算。

最後是記憶體被耗盡的案例，如 Figure 22：DFS 雖然較慢耗盡記憶體，卻也沒有再挖掘到更多區塊，而 BFS 和 MCTS 分別執行了 11000 次和 15000 次的 step 才耗盡了記憶體，MCTS 卻能取得最好的結果。而 Figure 23、24 更凸顯出這一點，當 DFS 和 BFS 都已經耗盡記憶體時，MCTS 卻能找到更多區塊。

從 4.3 節的實驗中可以看出一個共同的現象：通常 DFS 能夠執行最多次的 step，但搜尋的效率並不一定最好，大多數情況都是成長到一定的數字後就持平不再增加，直

到記憶體空間被耗盡；而 BFS 雖然有著穩定而緩慢的成長，甚至在少數案例中可以超越其他策略，但其缺點非常明顯，即記憶體使用率增加的非常快，以至於在所有的實驗中，BFS 所能執行的 step 次數幾乎都是墊底的；而 MCTS 在大部分的案例中都有較好的效率。



Chapter 6

Related Work

6.1 Path Selection Problem

在 [13][14] 中對 symbolic execution 所做的概述，提及了幾項目前的挑戰，其中一個重要的問題就是路徑選擇問題 (path selection problem)，當路徑爆炸問題已難以避免其發生，我們希望藉由有效的路徑選擇在資源限制內做最高效率的探索；目前已經提出的幾種解決方式如：KLEE[5] 中提出的深度優先搜尋法來優先探索最深的路徑，但很有可能被困在無限迴圈而進行了無效的探索。[2] 提出的 concolic testing，除了以 symbolic execution 執行，也會使用具體的輸入真正執行程式來蒐集執行路徑，[4] 就使用了這種作法。而 KLEE[5] 也支援隨機挑選路徑的方式來避免進行了無效的探索。

近期有加入統計和機率計算來改善 Path selection problem 的方法，如 [15]：傳統的 symbolic execution 只計算路徑的可滿足性 (satisfiability)，而無法得知路徑或程式碼的執行機率，它透過分析程式輸入計算路徑條件的解數量 (model counting)，進而計算出路徑的執行機率，讓使用者檢查計算出的機率和預想中的行為是否相同。而 [16] 則是利用隨機取樣技術 Bayesian estimation 和 hypothesis testing 於路徑選擇上，使 symbolic execution 會選擇相同性較低的路徑。[17] 則是設計了 fitness function 藉由路徑和分支的特徵給予價值高低，來引導 symbolic execution 在每次選擇路徑時的參考。

另外如 driller[18] 結合了 AFL[19] 的 fuzzing 技術，它將使用者輸入分類為需要特定值的特定輸入 (specific input) 和可接受各種數值的通用輸入 (general input)，並透過 symbolic execution engine 和 fuzzing engine 的切換來解決各自不擅長的部分。而 s2e[6] 則是利用選擇性的 symbolic execution，避免連同其他函式庫也一起分析，造成路徑數量大量增長。另外也有針對路徑成長，檢查 satisfiability 並做動態剪枝的方法 [20]。

6.2 Search-based test generation

search-based software testing (SBST) 是能夠根據原始碼來自動產生測試資料，通常使用基因演算法或其他人工智慧方法來實現，以提高測試對象的 code coverage，而其最重要的精神在於如何提供一個夠好的 fitness function 來決定何謂好的 test case。一個 SBST 工具軟體通常包含了：搜尋演算法、程式解的表示方法、程式解的修改方法、fitness function 評估何謂好的解；如同基因演算法是先給一組初始解，並給它一些修改的方法，如重排位元、調換位元、反轉位元等等，並將修改過的解以 fitness function 評估好壞，藉由不斷的重複找出好的解，已有的研究如 [21] 使用的是基因演算法，[22] 提出的使用模擬退火法，在這個領域也有套用 MCTS 類型演算法的研究，使用了 Nested monte carlo search 以產生測試資料的方法 [23][24]。

6.3 MCTS and Game AI

Monte Carlo 方法是一種隨機取樣的方法，在 1987 年由 Bruce Abramson 提出 [25]。而在 1989 年，Monte Carlo tree search 由 W. Ertel, J. Schumann 和 C. Suttner 提出，用來改善搜尋演算法的時間如 DFS、BFS 等等。而在 1993 年 B. Brügmann 首先將 Monte Carlo 方法用於圍棋上 [26]，直到 2006 年這個方法才由 Rémi Coulom 真正被命名為 Monte Carlo tree search[27]。之後 L. Kocsis and Cs. Szepesvári 以 MCTS 為基礎發表了 upper confidence bound 1 applied to trees (UCT) 演算法 [28]。在 2015 年由 Google Deepmind 研發的圍棋 AI AlphaGo[1]，使用 MCTS 和 deep learning 演算法，擊敗了人類職業選手，頓時之間 AI 和機器學習又成為電腦科學界的顯學。

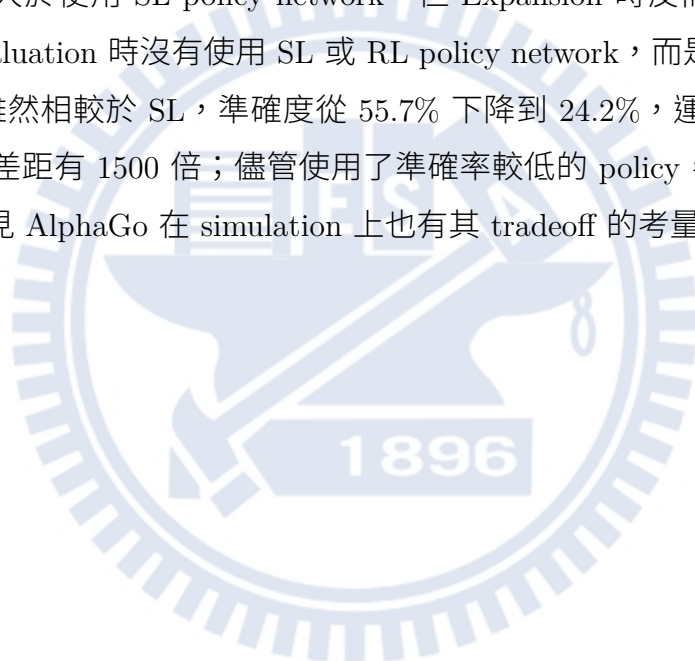
由於遊戲盤面的推估複雜度與其盤面可呈現的狀態成正比，當一個盤面有數百種可能下法時，人類通常只考慮其中幾種並做深入的考量，相對於電腦而言如果全部展開盤面並計算顯得十分緩慢，因而程式設計師開始利用各種方法嘗試縮小真正需要的搜尋的那幾種可能；由程式設計師針對不同遊戲寫不同規則來應對，是一件不可能的事情，所以無論是 MCTS 或機器學習，皆在於讓電腦能自己掌握哪些情況是必須要計算，哪些情況則沒有必要的。

6.4 MCTS in AlphaGo

AlphaGo 實作的搜尋演算法為 asynchronous policy and value MCTS algorithm(APV-MCTS)，並在演算法的不同階段整合類神經網路來提升效能。在 Selection 階段使用

的是 polynomial upper confidence trees 演算法 (PUCT, UCT 的變形), 來選擇要計算的子樹節點。由於圍棋中幾乎所有盤面都是 expandable 的, 所以 AlphaGo 設計了一個閥值, 當節點被走訪超過某個數值時, 才會產生新的子節點; 當新的節點要被產生時, 會先透過設計好的 *tree policy* 提供幾個較有可能的選擇, 再由 SL policy network 和 value network 來決定位置。Evaluation 階段則是由 value network 給出盤面價值, 同時也以 fast rollout policy 把盤面下完, 計算出贏家, 這兩個結果來共同決定最終的盤面價值。Backup 階段則將計算出的盤面價值往上更新這個子樹。

從 AlphaGo 的實驗結果可以看出, 演算法每個環節的參數都可能有影響, 如 SL policy network 和 value network 數值的混合比例、Expansion 時的 threshold、Selection 時的 Exploration constant。value network 的訓練也發現使用 RL policy network 來自我對奕的準確性大於使用 SL policy network, 但 Expansion 時反而使用 SL 的勝率會贏過 RL。在 Evaluation 時沒有使用 SL 或 RL policy network, 而是使用準確度較低的 rollout policy, 雖然相較於 SL, 準確度從 55.7% 下降到 24.2%, 運算速度卻能從 2 毫秒提升 3 微秒, 差距有 1500 倍; 儘管使用了準確率較低的 policy, 實驗卻發現整體上是比較好的, 顯見 AlphaGo 在 simulation 上也有其 tradeoff 的考量存在。



Chapter 7

Conclusions

7.1 conclusion

我們提出的演算法在實驗中證實，在相同的時間限制或資源限制下，比起傳統的 DFS 和 BFS 策略能有效的走訪更多程式碼，同樣在沒有修改任何程式或資料的情況下，我們額外使用一棵樹來記錄路徑的資訊和親子關係，就算沒有使用 MCTS 的方法，紀錄路徑間的親子關係也有可能用來作為其他演算法上的 cut 或 pruning 使用；唯一被修改的是路徑進入 symbolic execution engine 計算的順序，這使我們的方法與其他技術如 concolic testing[2], veritesting[29], driller[18] 整合上相較容易。

但我們的演算法仍然可能會有以下的問題，第一是路徑爆炸問題，如果我們完全不修剪路徑，它仍然可能發生，只能盡力在發生前挑選價值高的路徑進行搜索；如果借助我們建立的樹來進行修剪，雖然可以延緩甚至避免這個問題發生，但有些路徑可能就不會被探索到。第二是 symbolic execution 既有的問題，當它遇到大量迴圈如 strcmp 函式，會產生大量路徑，而且很難找到能夠離開的路徑，我們的演算法也會遇到這個問題，在部分的實驗中顯示了仍然有可能被困住而沒有繼續增加覆蓋率，這只能依靠結合其他技術來獲得解決。最後是 CFG 的問題，當靜態分析發生錯誤，如產生出的 CFG 有部分不正確，或是被混淆代碼 (obfuscated code) 等等可能的因素，那演算法對於模擬計算出的期望值就可能沒有效果，這時就只能依靠路徑的已執程式碼區塊數量來判斷，演算法的效果會變差。

7.2 further work

在這篇論文中我們提出了以 MCTS 的角度來作為 symbolic execution 的 search strategy，以往的作法多是將路徑視為集合，會在所有的已知路徑中進行選擇，但我們認為以樹的角度來嘗試解決這個問題可能會有更好的結果；因此我們建立了一棵樹用來記錄所有路徑間的親子關係，就可以嘗試套用 tree optimization 的做法來套用到我們的架構上。

從 AlphaGo 中可以看出，要結合機器學習的方法，MCTS 是一個很好的嘗試方向，藉由一個預先設計好的 tree search procedure 來進行搜尋，並從中改善它，藉由引導其方向可能有更好的效能。在我們的架構中，Selection 只評估了能增加多少覆蓋率和路徑深度，但已有一些研究藉由計算路徑的執行機率或 fitness function 來讓選擇更加精準，目前的演算法使用的是較輕量級的價值評估函式，並沒有運用太多的 domain knowledge，如果能結合對敏感函式或危險行為的偵測或前述的方法，在漏洞尋找上相信能有更大的成效。Simulation 目前使用了 CFG 來判斷路徑接下來的執行流程，但並沒有根據路徑目前的狀態和 satisfiability 進行選擇，如果用其他的作法如 model checking 甚至 concrete execution 或許可以在模擬上更加精準，但相對的也會花費更多時間。

我們實作採用的框架屬於較易於開發但小型的平台，如能移植到較大且完整的系統如 s2e 中，應能支援更大且複雜的程式。

References

- [1] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587 (2016), pp. 484–489.
- [2] Koushik Sen. “Concolic testing”. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM. 2007, pp. 571–572.
- [3] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: directed automated random testing”. In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 213–223.
- [4] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: a concolic unit testing engine for C”. In: *ACM SIGSOFT Software Engineering Notes*. Vol. 30. 5. ACM. 2005, pp. 263–272.
- [5] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.” In: *OSDI*. Vol. 8. 2008, pp. 209–224.
- [6] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. “The S2E platform: Design, implementation, and applications”. In: *ACM Transactions on Computer Systems (TOCS)* 30.1 (2012), p. 2.
- [7] Sang Kil Cha et al. “Unleashing mayhem on binary code”. In: *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 380–394.
- [8] Julien Vanegue, Sean Heelan, and Rolf Rolles. “SMT Solvers in Software Security.” In: *WOOT*. 2012, pp. 85–96.
- [9] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on*

- Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24. URL: http://dx.doi.org/10.1007/978-3-540-78800-3_24.
- [10] Cameron B Browne et al. “A survey of monte carlo tree search methods”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43.
 - [11] Jeff Bradberry. *Introduction to Monte Carlo Tree Search*. URL: <https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/>.
 - [12] Yan Shoshitaishvili et al. “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *IEEE Symposium on Security and Privacy*. 2016.
 - [13] Asankhaya Sharma. *A critical review of dynamic taint analysis and forward symbolic execution*. Tech. rep. Technical report, 2012.
 - [14] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)”. In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 317–331.
 - [15] Jaco Geldenhuys, Matthew B Dwyer, and Willem Visser. “Probabilistic symbolic execution”. In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM. 2012, pp. 166–176.
 - [16] Antonio Filieri et al. “Statistical Symbolic Execution with Informed Sampling”. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 437–448. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635899. URL: <http://doi.acm.org/10.1145/2635868.2635899>.
 - [17] Tao Xie et al. “Fitness-guided path exploration in dynamic symbolic execution”. In: *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009*. 2009, pp. 359–368. ISBN: 9781424444212. DOI: 10.1109/DSN.2009.5270315.

- [18] Nick Stephens et al. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution”. In: *Proceedings of the Network and Distributed System Security Symposium*. 2016.
- [19] Michal Zalewski. *American Fuzzy Lop*. URL: <http://lcamtuf.coredump.cx/afl>.
- [20] 陳盈伸. “符號執行之動態路徑修剪技術”. 英文. MA thesis. 臺灣大學, 2016, pp. 1–49. DOI: 10.6342/NTU201602958.
- [21] S Xanthakis et al. “Application of genetic algorithms to software testing”. In: *Proceedings of the 5th International Conference on Software Engineering and Applications*. 1992, pp. 625–636.
- [22] Salah Bouktif, Houari Sahraoui, and Giuliano Antoniol. “Simulated annealing for improving software quality prediction”. In: *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM. 2006, pp. 1893–1900.
- [23] Robert Feldt and Simon Poulding. “Broadening the search in search-based software testing: it need not be evolutionary”. In: *Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th International Workshop on*. IEEE. 2015, pp. 1–7.
- [24] Simon Poulding and Robert Feldt. “Generating structured test data with specific properties using Nested Monte-Carlo Search”. In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM. 2014, pp. 1279–1286.
- [25] “Front Matter”. In: *The Expected-Outcome Model of Two-Player Games*. Ed. by Bruce Abramson. Research Notes in Artificial Intelligence. Morgan Kaufmann, 1991, pp. iii –. DOI: <http://dx.doi.org/10.1016/B978-0-273-03334-9.50001-2>. URL: <http://www.sciencedirect.com/science/article/pii/B9780273033349500012>.
- [26] Bernd Bruegmann. *Monte Carlo Go*. Tech. rep. Technical report, 1993.
- [27] Rémi Coulom. “Efficient selectivity and backup operators in Monte-Carlo tree search”. In: *In: Proceedings Computers and Games 2006*. Springer-Verlag, 2006.
- [28] Levente Kocsis and Csaba Szepesvári. “Bandit based Monte-Carlo Planning”. In: *In: ECML-06. Number 4212 in LNCS*. Springer, 2006, pp. 282–293.

- [29] Thanassis Avgerinos et al. “Enhancing Symbolic Execution with Veritesting”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 1083–1094. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568293. URL: <http://doi.acm.org/10.1145/2568225.2568293>.



Appendix A

實驗數據

下列表格為 4.3 節圖表的數據，其中 round 表示執行為第幾次 step 時，表格內數字為 basic blocks 涵蓋數量，最下方一行則是終止時的 round 數和 basic blocks 涵蓋數量。

Table 4: cpp-markdown & cp 長時間執行數據

round	cpp-markdown			cp		
	BFS	DFS	MCTS	BFS	DFS	MCTS
100	47	47	47	24	47	38
500	196	205	199	50	164	119
1000	385	376	349	92	178	202
2000	459	497	477	126	186	331
5000	519	572	541	196	222	462
10000	560	579	559	215	222	573
20000	629	579	593	226	222	N/A
	102k/661	110k/588	143k/606	20k/226	64k/222	17k/581

Table 5: echo & gif2png 長時間執行數據

round	echo			gif2png		
	BFS	DFS	MCTS	BFS	DFS	MCTS
100	27	34	37	23	14	26
500	43	52	67	49	34	70
1000	57	57	71	53	64	96
2000	77	71	79	58	64	107
5000	105	84	92	61	64	107
10000	118	87	93	63	64	107
20000	129	87	102	73	64	107
	21k/129	94k/87	26k/103	24k/74	64k/64	79k/107

Table 6: hostname & ls 長時間執行數據

round	hostname			ls		
	BFS	DFS	MCTS	BFS	DFS	MCTS
100	19	35	25	38	44	35
500	57	61	95	74	125	128
1000	60	61	128	98	156	169
2000	61	61	146	133	158	241
5000	63	61	184	185	198	344
10000	64	61	184	201	198	493
20000	65	61	184	N/A	198	646
	25k/65	119k/61	65k/184	12k/209	78k/198	21k/652

Table 7: mkdir & ps 長時間執行數據

round	mkdir			ps		
	BFS	DFS	MCTS	BFS	DFS	MCTS
100	24	49	40	32	27	29
500	76	71	96	55	118	72
1000	95	79	97	84	185	83
2000	110	82	120	107	185	96
5000	126	82	389	116	185	107
10000	133	82	532	128	185	271
20000	154	82	568	N/A	185	N/A
	34k/171	156k/82	35k/572	11k/131	91k/185	15k/431

Table 8: readelf & touch 長時間執行數據

round	readelf			touch		
	BFS	DFS	MCTS	BFS	DFS	MCTS
100	39	41	43	24	48	39
500	95	124	103	71	117	141
1000	113	234	163	91	161	218
2000	133	323	280	116	184	286
5000	177	393	352	142	222	307
10000	185	415	421	161	222	334
20000	201	464	456	175	222	339
	22k/205	35k/466	90k/652	24k/182	65k/222	139k/340