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不停成長，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模擬較有價值的路徑，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、廣度優先搜尋的效率。

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

English Abstract

Keywords : Hello Keywords

Contents

List of Figures

List of Tables

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，造成探索路徑需要花費巨大的運算資源，這個問題被稱為 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(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；此演算法主要的精神在於對一棵樹，選擇子結點並嘗試用模擬的方式估計該結點的價值。和 symbolic execution 相當類似的地方在於同為需要探索一棵樹，而且要避免探索一些我們不感興趣或是沒有必要探索的路徑，例如無窮迴圈的狀況；我們認為結合蒙地卡羅樹搜尋做為 symbolic execution 挑選路徑執行的演算法，相較於傳統的深度優先搜尋法或廣度優先搜尋法，能有較佳的搜尋效率。

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流程和優點。

2.1 Symbolic execution

在本篇論文中所提及的 Symbolic execution 為 Dynamic symbolic execution(又名為 Concolic execution)，首先由 K. Sen[1] 提出，和基於其想法實作的程式 DART[2]、CUTE[3]，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[4] 和不須程式碼而直接分析程式執行檔的 binary-based symbolic execution 如 S2E[5]、Mayhem[6]；symbolic execution engines 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[7]、Z3[8] 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。

2.2 Path Exploration

雖然 Symbolic execution 理論上能夠探索所有執行路徑，但在探索的過程中可能會遭遇到一些問題：當分支條件取決於 symbolic 變數時，很有可能兩邊的條件都有可能

成立，這意謂著程式必須複製並分別維護兩條路徑，直到發現該路徑無解為止，當路徑被不斷的複製就產生了路徑爆炸問題 (path explosion problem)。

2.3 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 [9] 中說明了 MCTS 演算法的基本流程，如 Figure 1：

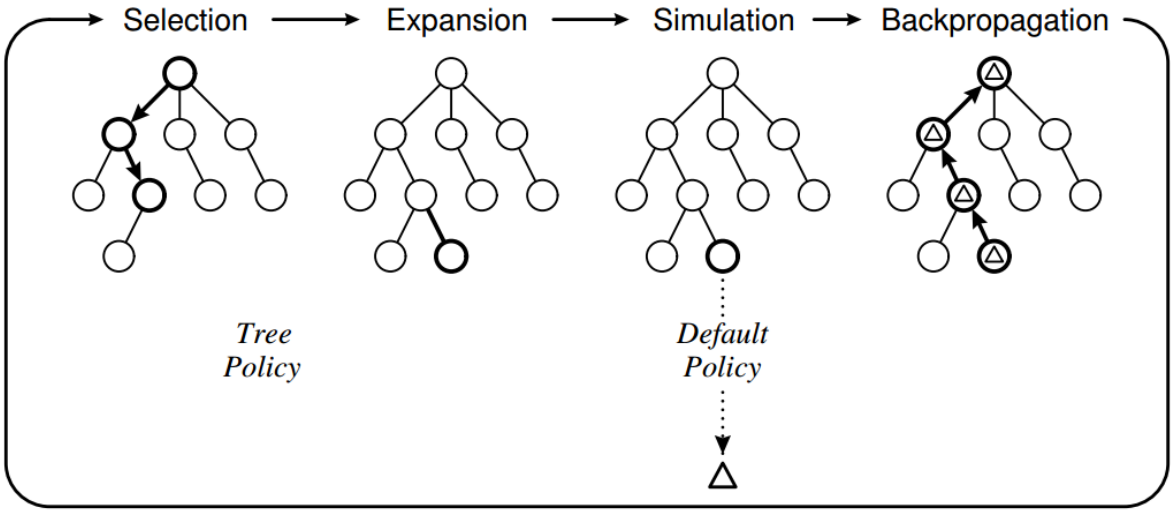

Figure 1: Monte Carlo Tree Search

- **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 1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 1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。
- **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 [10] 便指出，Tree Policy 可以使用 upper confidence bound (UCB) 演算法來取代隨機挑選，當可能的選擇有很多，卻只有少數一兩個有可能被選擇時，隨機選擇不太容易挑到，而造成模擬的結果不精準，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

Design

MCTS 演算法是為了要在特定時間或電腦資源使用量內做出某個決策，利用隨機模擬與統計的方式尋找出一個最佳解，如何最有效的擴展空間狀態樹是我們的演算法可以仿效之處。對 symbolic execution 而言，目標就變成如何達到更高的執行覆蓋率，以盡量的挖掘出可能的漏洞。

我們希望在套用 MCTS 演算法後，能延緩 path explosion problem 的發生，並使 symbolic execution 相較於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。

Algorithm 1 applying UCT algorithm to symbolic execution

```
1: function SEARCH(p_r)
2:     set p_r as root of Tree T
3:     B ← ∅
4:     while within computational budget do
5:         p ← TreePolicy(T)
6:         B ← B ⋃ p
7:         S ← step(p)
8:         for each path p_c ∈ S do
9:             V ← DefaultPolicy(p_c)
10:            Q(p_c) ← α|V−B|/N + β|p_c|
11:            add a new child p_c to p
12:        end for
13:        BackPropagation(p)
14:    end while
15: end function
```

Algorithm 1 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 *TreePolicy* 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 engine 進行計算，將這條路徑 step 一步；step 後可能產生數條路徑，我們以集合 S 來表示，S 中的路徑們代表 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 $DefaultPolicy$ 來模擬其未來可能的走向，並評估此路徑的價值 $Q(p_c)$，並將 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 $DefaultPolicy$ 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($DefaultPolicy$ 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，$TreePolicy$ 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 2 是 Algorithm 1 中所提及的函式實作部分。$TreePolicy$ 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 中常被提及的特徵：expolitation 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。$DefaultPolicy$ 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。$BackPropagation$ 則是更新兩項數值，$N(v)$ 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 2 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
      $n \leftarrow BestChild(n, C)$
    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 $j = 1$; ($j < M$)and(v 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)
  $Q_{max} \leftarrow \arg\max_{p_c \in p} Q(p_c)$

$$\textbf{return} \arg\max_{p_c \in p} \frac{Q(p_c)}{Q_{max}} + C\sqrt{\frac{2\ln N(p)}{N(p_c)}}$$

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

Chapter 4

Evaluation

4.1 實驗環境與方法

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) [11] 上，並在 Ubuntu 16.04 作業系統上安裝與執行所有需要的程式和測試的目標程式，目標程式的名稱與版本如下 Table. 4.1。硬體環境使用的是 Intel i7-2600k 處理器以及 24 GB 記憶體。

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

4.2 常數對演算法的效率影響

4.3 演算法與其他方法的比較

4.4 長時間執行的效率比較

Chapter 5

Related Work

5.1 Path Selection Problem

在 [12][13] 中對 symbolic execution 所做的概述，提及了幾項目前的挑戰，其中最大的問題就是路徑選擇問題 (path selection problem)，當路徑爆炸問題已成一個不可避免的問題，目前已經提出的幾種解決方式如：KLEE[4] 中提出的深度優先搜尋法來優先探索最深的路徑，中便使用這種作法，但很有可能被困在無限迴圈而進行了無效的探索。[1] 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，[3] 就使用了這種作法。而 KLEE[4] 也支援隨機挑選路徑的方式來避免進行了無效的探索。

另外如 driller[14] 結合了 AFL[15] 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[5] 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，檢查其可滿足性 (satisfiability) 並做動態剪枝的方法 [16]。

5.2 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 [17]。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 [18]，直到 2006 年這個方法才由 Rémi Coulom 真正被

命名為 Monte Carlo tree search[19]。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 [20]。在 2015 年由 Google Deepmind 研發的圍棋 AI AlphaGo[21]，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。

Chapter 6

Conclusion

我們提出的演算法在實驗中證實，在相同的時間限制和資源限制下，比起傳統的 DFS 和 BFS 策略能有效的探訪更多程式碼，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 engine 計算的順序，這使我們的方法與其他技術如 concolic testing[1], veritesting[22], driller[14] 整合的難度有可能是變低的。

但我們的演算法仍然可能會有以下的問題，第一是 path explosion problem，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，我們的演算法也會遇到這個問題，這只能依靠 veritesting 或 fuzzing 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行程式碼區塊數量來判斷，演算法的效果會變差。

References

[1] Koushik Sen. "Concolic testing". In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM. 2007, pp. 571–572.

[2] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing". In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 213–223.

[3] 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.

[4] 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.

[5] 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.

[6] Sang Kil Cha et al. "Unleashing mayhem on binary code". In: *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 380–394.

[7] Julien Vanegue, Sean Heelan, and Rolf Rolles. "SMT Solvers in Software Security." In: *WOOT*. 2012, pp. 85–96.

[8] 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.

[9] 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.

[10] Jeff Bradberry. *Introduction to Monte Carlo Tree Search*. URL: https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/.

[11] Yan Shoshitaishvili et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy*. 2016.

[12] Asankhaya Sharma. *A critical review of dynamic taint analysis and forward symbolic execution*. Tech. rep. Technical report, 2012.

[13] 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.

[14] Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution". In: *Proceedings of the Network and Distributed System Security Symposium*. 2016.

[15] Michal Zalewski. *American Fuzzy Lop*. URL: http://lcamtuf.coredump.cx/afl.

[16] 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.

[17] "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.

[18] Bernd Bruegmann. *Monte Carlo Go*. Tech. rep. Technical report, 1993.

[19] Rémi Coulom. "Efficient selectivity and backup operators in Monte-Carlo tree search". In: *In: Proceedings Computers and Games 2006*. Springer-Verlag, 2006.

[20] Levente Kocsis and Csaba Szepesvári. "Bandit based Monte-Carlo Planning". In: *In: ECML-06. Number 4212 in LNCS*. Springer, 2006, pp. 282–293.

[21] Google Research Blog. *AlphaGo: Mastering the ancient game of Go with Machine Learning*. URL: https://research.googleblog.com/2016/01/alphago-mastering-ancient-game-of-go.html.

[22] 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.