Guiding Symbolic Execution with A-star

Theo De Castro Pinto[1,2], Antoine Rollet[1], Grégoire Sutre[1], and Ireneusz Tobor[2]

[1]Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400, Talence, France
{theo.de-castro-pinto, antoine.rollet, gregoire.sutre}@labri.fr
[2]Serma Safety & Security, F-33600, Pessac, France
{t.de-castro, i.tobor}@serma.com

Abstract

Symbolic execution is widely used to detect vulnerabilities in software. The idea is to symbolically execute the program in order to find an executable path to a target instruction. For the analysis to be fully accurate, it must be performed on the binary code, which makes the well-known issue of state explosion even more critical. In this paper, we introduce a novel exploration strategy for symbolic execution aiming to limit the number of explored paths. Our strategy is inspired from the A* algorithm and steered towards least-explored parts of the program. We compare our approach, using the Binsec tool, to three other classical strategies: depth-first (DFS), breadth-first (BFS) and non-uniform random (NURS). Our experiments on real size programs show that our approach is promising.

Based on the paper [4] presented at the 21st International Conference on Software Engineering and Formal Methods (SEFM 2023).

1 Introduction

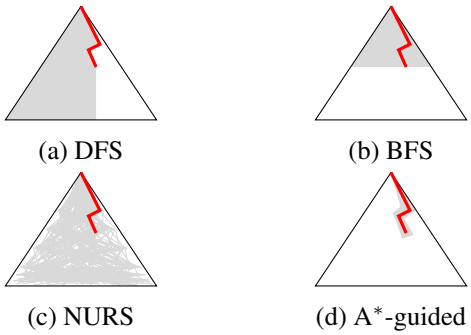
Context. Software verification is a crucial step during the development of programs permitting to discover potential failures. It consists not only in assessing the correct behavior of the program but also in checking if vulnerabilities exist. The number of inputs of a program is usually very big, inducing a huge number of possible paths. A popular technique used to handle this problem is symbolic execution [7]. A major problem of this approach is that it generally does not scale well on real size programs. The order of exploration is crucial and decided by the exploration strategy, which can be for instance depth-first (DFS), breadth-first (BFS) or non-uniform random (NURS).

Contributions. In this paper, we introduce two novel exploration strategies for symbolic execution, inspired by the well-known A* algorithm [6]. We first adapt the A* algorithm to symbolic execution of binary code, using a precomputed distance heuristic, which has never been done previously to our knowledge. We then improve this basic A*-like strategy to steer the exploration towards least-explored parts of the program. The total number of explored paths is reduced, implying better performance. Our strategies have been implemented in the binary code analysis tool Binsec [5]. We present an experimental evaluation of our two A*-like exploration strategies on seven programs, two of them being of real size (Wookey's bootloader [1] and the NetBSD `leave` command). Our experiments show that our approach is promising.

Related Work. In 2021, Blondin et al. proposed an approach based on the A* algorithm [6] to perform reachability analysis on Petri nets [3]. Their results showed that using this approach outperforms existing state-of-the-art Petri nets tools. The idea is to use distance oracles to guide the exploration of Petri nets. Regarding symbolic execution, many strategies aiming to guide the exploration towards more promising paths have been proposed in the literature. Some of them prioritize paths that are closer to the target state [2, 9] while others prioritize paths that explore new parts of the program [8, 11]. In both cases, only partial aspects of the A* algorithm are implemented. To our knowledge, none of them apply both strategies, and they are applied on source code. Our proposal combines both of these concepts into a novel exploration strategy, and applies it directly on binary code.

```
1  #define MAX_SIZE 10000000
2  #define EXPECTED_SIZE 100
3  void valid(int y) {
4      int x;
5      for (x = 0; x < MAX_SIZE; x++) {
6          if (!correct(y)) break;
7          y--;
8      }
9      if (x != EXPECTED_SIZE) trap();
10     critical();
11 }
```

Listing 1: C-style running example.



(a) DFS (b) BFS (c) NURS (d) A*-guided

Figure 1: Illustration of different symbolic execution strategies.

2 Running Example

The code given in Listing 1 is a simplified version of a security-critical code inspired from a real-life application. The parameter `y` of the function `valid` is a secret value that an attacker is not supposed to know. This value must satisfy a certain condition, namely that `correct`(n) returns `true` for all integers n with $y - 99 \leq n \leq y$, and `correct`$(y - 100)$ returns `false` (where y is the initial value of the parameter `y`, i.e., y is the actual argument). Note that the corresponding loop (lines 5–8) may, in fact, be traversed up to 10^7 times. If the above-mentioned condition on `y` is satisfied then the `critical` function is executed, otherwise a counter-measure, here `trap`, is triggered. Our goal is to use symbolic execution to (efficiently) find an executable path from the start of the `valid` function to the target `critical` function.

A depth-first (DFS) strategy either exits the loop early and ends up in the `trap` function, or executes the loop entirely and still ends up in the `trap` function. This behavior is illustrated in Figure 1a, where the red branch is the only one leading to the target, and the gray zone represents the branches already explored. A breadth-first (BFS) strategy is also highly inefficient as it generates all branches of length smaller than the length of the branch reaching the target, including the ones that are stuck in the `trap` function. Its behavior is exhibited in Figure 1b where a large part of the reachability tree is explored. A non-uniform random (NURS) strategy chooses randomly which branch to explore further (see Figure 1c). This strategy can be lucky but it mostly fails on real size programs.

The approach proposed in this paper is inspired from the A* algorithm and aims to explore a limited amount of branches. The resulting exploration strategy is illustrated in Figure 1d, where only a very small portion of the whole tree is explored.

3 Symbolic Execution Based on A*

Symbolic execution has originally been proposed for program testing [7], but the technique can also be used for reachability analysis. Our main contribution concerns exploration strategies for symbolic execution, which uses a worklist to order path exploration. An element of the worklist is a node of the exploration tree presented in Figure 1 and an associated priority. Elements of the worklist are addressed from the lowest to highest priority. Hence, the order of exploration can be customized via a priority function `Prio`. Naturally, the classical search exploration strategies DFS, BFS and NURS can be encoded as priorities. Given a node u to add to a worklist W, the priority functions are given by:

$$\texttt{PrioDFS}(u, W) = \begin{cases} 0 & \text{if } W = \emptyset \\ \min\{n.\texttt{priority} \mid n \in W\} - 1 & \text{otherwise} \end{cases}$$

$$\texttt{PrioBFS}(u, W) = \begin{cases} 0 & \text{if } W = \emptyset \\ \max\{n.\texttt{priority} \mid n \in W\} + 1 & \text{otherwise} \end{cases}$$

$$\texttt{PrioNURS}(u, W) = \text{random}(0, 1)$$

Recall that A* is a single-pair shortest path algorithm for nonnegatively weighted directed graphs. Assume that we are given such a graph together with a source vertex and a target vertex. Let V denote the set of vertices of the graph. The main idea of the A* algorithm is to guide the exploration using a heuristic function $h : V \to \mathbb{N} \cup \{+\infty\}$ that underestimates the (minimal) distance from any vertex to the target vertex. Note that $h(v)$ may be $+\infty$ if there is no path from v to the target vertex. When A* picks a vertex to process from its worklist, it chooses a vertex v that minimizes the sum $g(v) + h(v)$, where $g(v)$ is the weight of the shortest path *seen so far* from the source vertex to v.

We adapt A* to symbolic execution as follows. The g value of a node u is the length of the explored branch leading to u. Note that this is an underapproximation of the real g value used in the A* algorithm: ideally, the length of the shortest branch to the same symbolic state as u should be used. The distance from an instruction ℓ to a target instruction t is the length of the shortest executable path from ℓ to t. We compute an underapproximation h of this distance that only accounts for the stack contents and ignores the values of the variables. To compute the h values, we first generate a summary of every procedure in the program, and then apply Dijkstra's algorithm starting from the target instruction (a more efficient algorithm is proposed in [2]). With u a node to add to a worklist W and $u.instr$ the current instruction we obtain:

$$\texttt{PrioASTAR}(u, W) \;=\; g(u) + h(u.instr) \tag{1}$$

The issue with the approach presented previously is that in large programs the g value tends to dominate the h value. Consequently, the strategy mostly behaves like a BFS which does not scale well. To fix this issue, we propose to replace g by another measure that still accounts for the length of the branch from the root of the tree to u, but prioritizes nodes corresponding to parts of the system that have rarely been visited. The resulting priority function is defined as:

$$\texttt{PrioASTAR-2}(u, W) \;=\; g'(u) \cdot \lambda(\mu(u)) + h(u) \tag{2}$$

Where u is a node to add to a worklist W. We define $g'(u)$ as the elementary depth of the branch, that is the number of unique instructions visited along the branch until the current instruction was first visited. And $\mu(u)$ is defined as the number of times the current instruction has been visited along the branch. Finally, λ is an attenuation function to mitigate the impact of the g' values, it may for example be a logarithmic function.

4 Experimental Results

We evaluate our new approach on seven programs: the running example in Listing 1, a complex version of it exhibiting the limitations of the first priority function based on A*, three "*crackme*" challenges [10] which are relatively easy to solve and with a reasonable size (around 200 instructions), and two "real size" programs namely the Wookey bootloader [1] which is a popular software designed by the ANSSI[1] meant to be robust against various type of attacks (∼10K locations), and the `leave` command of NetBSD (∼100K locations). We use the symbolic execution tool Binsec (version 0.6), in which we have implemented our new strategies. A time limit of 100 seconds is allowed for each experiment, beyond which we stop it and report a timeout. The NURS exploration strategy has been run ten times on each program and the mean of the results are displayed. Timeouts for this strategy are discarded in order to not affect the final results. Results are shown in Figure 2.

Clearly, our new exploration strategy `astar-2` always outperforms the classical strategies. Moreover, it also always outperforms the strategy solely based on `astar`, as expected. The `astar` exploration strategy is generally not powerful enough to reach the target on real programs (`leave`, Wookey's bootloader). Regarding the duration of the symbolic execution, the strategy `astar-2` also always outperforms the other strategies. Note that the number of unrolled instructions is not directly correlated to the execution time of symbolic execution. In fact, what really slows it down are satisfiability queries, which are made at conditional branching points.

[1] French National Cybersecurity Agency.

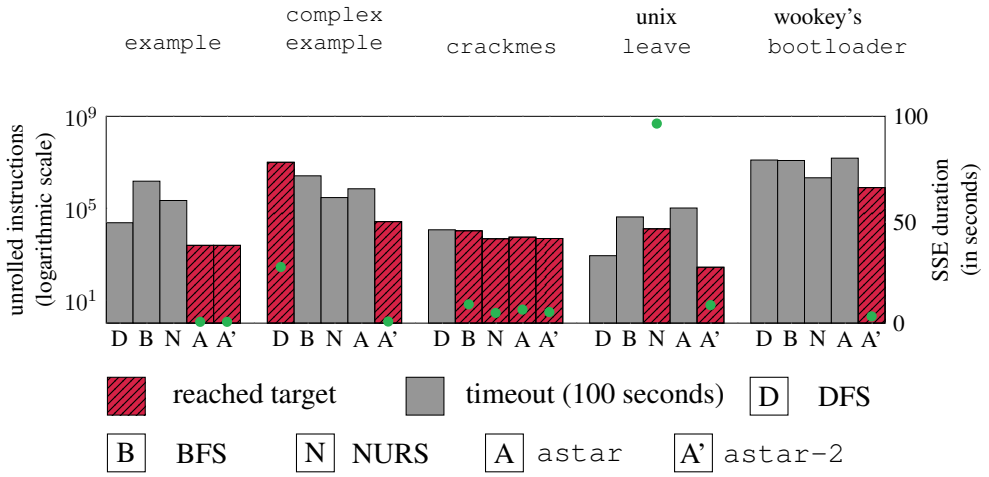


Figure 2: Experimental results obtained with Binsec: bars represent the number of unrolled instructions in a logarithmic scale and dots represent the SSE duration in seconds.

5 Conclusion

In this paper, we have introduced a novel exploration strategy for symbolic execution inspired from the A* algorithm permitting to find efficiently an executable path to a target instruction. This approach orders the exploration of symbolic states by using heuristics permitting to visit in priority states that have been less explored. Consequently the number of paths to explore is smaller than in usual approaches such as DFS, BFS and NURS, implying better performance.

References

[1] ANSSI: Wookey. `https://wookey-project.github.io/` (2018)

[2] Babić, D., Martignoni, L., McCamant, S., Song, D.: Statically-directed dynamic automated test generation. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. pp. 12–22 (2011)

[3] Blondin, M., Haase, C., Offtermatt, P.: Directed reachability for infinite-state systems. In: Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part II 27. pp. 3–23. Springer (2021)

[4] De Castro Pinto, T., Rollet, A., Sutre, G., Tobor, I.: Guiding symbolic execution with a-star. In: International Conference on Software Engineering and Formal Methods. pp. 47–65. Springer (2023)

[5] Djoudi, A., Bardin, S.: Binsec: Binary code analysis with low-level regions. In: Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21. pp. 212–217. Springer (2015)

[6] Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics **4**(2), 100–107 (1968)

[7] King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976)

[8] Li, Y., Su, Z., Wang, L., Li, X.: Steering symbolic execution to less traveled paths. ACM SigPlan Notices **48**(10), 19–32 (2013)

[9] Ma, K.K., Yit Phang, K., Foster, J.S., Hicks, M.: Directed symbolic execution. In: Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18. pp. 95–111. Springer (2011)

[10] NoraCodes: crackmes. `https://github.com/NoraCodes/crackmes/` (2017)

[11] Xie, T., Tillmann, N., De Halleux, J., Schulte, W.: Fitness-guided path exploration in dynamic symbolic execution. In: 2009 IEEE/IFIP International Conference on Dependable Systems & Networks. pp. 359–368. IEEE (2009)