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.

Keywords: Symbolic execution · Program analysis · Binary code analysis · A* algorithm.

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. Software verification techniques include (automatic) formal proofs [15], testing [5], fuzzing [16], code review and program analysis [3, 6–8, 14]. This paper deals with program analysis of binary code, more precisely with the problem of efficiently finding an executable path to a target instruction (aka the line reachability problem). 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 [14]. It is an exploration technique aiming to find inputs of a program, with the help of a constraint solver, corresponding to a target path of the program. More precisely, considering a target path π of the program, a corresponding path predicate formula representing the constraints over the input variables along π is sent to a constraint solver. If the formula is satisfiable, then the path is executable, and a solution of the constraint system corresponds to a possible input set of the program activating π. 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). In this work, we consider binary code. Directly analyzing the binary code is necessary to verify that the compilation did not introduce new behaviors or vulnerabilities, but it is challenging. This stems from the fact that a lot of information is lost after the compilation and that binary code contains a lot more instructions than source code.

Contributions. In this paper, we introduce two novel exploration strategies for symbolic execution, inspired by the well-known A* algorithm [13]. A* is an efficient single-pair shortest path algorithm, therefore using it in order to quickly reach a target during symbolic execution makes sense. This key insight is at the core of Blondin et al.'s efficient explicit reachability analysis tool for Petri nets [4]. 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.

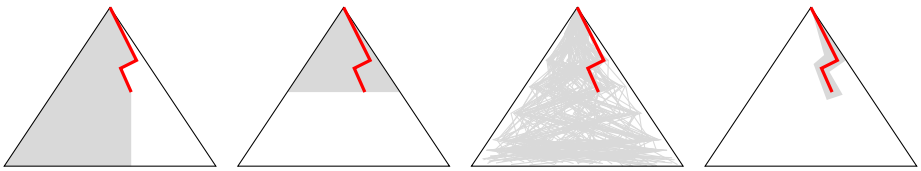
We provide a formal description of our approach on transition systems, which makes it generic and then applicable in various contexts. Our strategies have been implemented in the binary code analysis tool Binsec, although dynamic jumps are not currently handled. We present an experimental evaluation 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. A replication package is available at Zenodo [10].

Related Work. Symbolic execution [14] is a powerful technique to analyze programs. It is used in many program analysis tools, for instance KLEE [5], MIASM [19], ANGR [24] and Binsec [11]. KLEE is a dynamic symbolic execution engine that is used on source code (translated to LLVM). MIASM and ANGR are binary analysis platforms that combine both static and dynamic symbolic execution. Binsec is a framework for binary code analysis based on formal approaches such as symbolic execution, abstract interpretation [8], SMT solving [9] and fuzzing [16]. The exploration strategies provided by Binsec are BFS, DFS and NURS. Common uses of symbolic execution include test case generation [5], input generation for fuzzing [25] or even vulnerability detection [12, 23].

In 2021, Blondin et al. proposed an approach based on the A* algorithm [13] to perform reachability analysis on Petri nets [4]. 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. Our approach generalizes this concept to any labeled transition system. We also propose some enhancements in order to reach targets more efficiently in real programs. Many strategies aiming to guide the exploration towards more promising paths have

```
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 1.1: C-style running example.



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

Fig. 1: Illustration of different symbolic execution strategies.

been proposed in the literature. Some of them prioritize paths that are closer to the target [2, 18] while others prioritize paths that explore new parts of the program [17, 26]. 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.

2 Running Example

The code given in Listing 1.1 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`. 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. For our discussion, the contents of these two functions does not matter, except that the `trap` function is an infinite loop whose body contains two small paths (corresponding to security measures). 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.

Let us look qualitatively at the behavior of symbolic execution on this example regarding different exploration strategies. 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. In both cases, it is highly inefficient, as a huge number of branches are explored in the `trap` function before the loop exits with the expected value of 100 for `x`. 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 lesser 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). Again, because of the `trap` function, a huge number of branches are generated on average before reaching the target. 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. A more precise comparison of these four exploration strategies on this example will be given at the end of the next section.

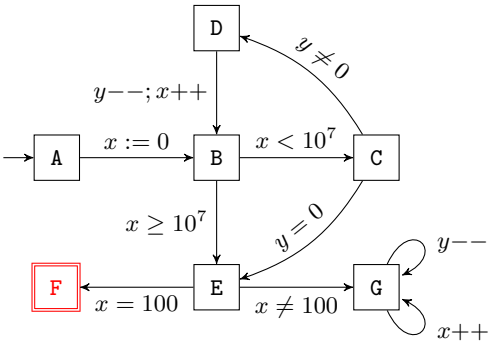
3 A* Guided Symbolic Execution

Many verification questions, including vulnerability detection, can be phrased as reachability queries over a labeled transition system providing the operational semantics of the system under analysis. We start by recalling a few preliminary notions on reachability in labeled transition systems. The remainder of the section focuses on symbolic execution and discusses various exploration strategies.

Reachability in Labeled Transition Systems. A (non-deterministic) *labeled transition system* is a 5-tuple $\mathcal{S} = (C, \Sigma, \rightarrow, I, F)$ where C is a possibly infinite set of *configurations*, Σ is a finite set of *actions*, $\rightarrow \subseteq C \times \Sigma \times C$ is a *labeled transition relation*, $I \subseteq C$ is a set of *initial configurations*, and $F \subseteq C$ is a set of *final configurations*. A *run* in $\mathcal{S}$ is an alternating sequence $\rho = (c_0, a_1, c_1, \ldots, a_n, c_n)$ of configurations $c_i \in C$ and actions $a_i \in \Sigma$ such that $c_{i-1} \xrightarrow{a_i} c_i$ for all i. We say that ρ is a run *from* c_0 *to* c_n and we write $\rho = c_0 \xrightarrow{a_1} c_1 \cdots \xrightarrow{a_n} c_n$. The word $a_1 \cdots a_n$ is called the *trace* of ρ. Given two configurations $c, c' \in C$ and a word $w \in \Sigma^*$, the notation $c \xrightarrow{w} c'$ means that there exists[3] a run from c to c' with trace w. The length of w is denoted by $|w|$. We say that c' is *reachable* from c, written $c \xrightarrow{*} c'$, when $c \xrightarrow{w} c'$ for some $w \in \Sigma^*$.

Our main objective is to determine whether there exists a run from an initial configuration to a final configuration. Formally, the *reachability problem* asks, given a LTS $\mathcal{S} = (C, \Sigma, \rightarrow, I, F)$, whether there exists $c \in I$ and $c' \in F$ such

[3] Due to non-determinism, there may be several runs from c to c' with trace w.



(a) States and transitions of the counter machine.

location	h(location)
A	3
B	2
C	2
D	3
E	1
F	0
G	$+\infty$

(b) Estimated distances to the final location F (so-called h values).

Fig. 2: Counter machine corresponding to the inlined code given in Listing 1.1, assuming that the `correct` function simply checks that its argument is nonzero.

that $c \xrightarrow{*} c'$. In theory, the reachability problem is only a decision problem. But, in practice, a trace $w \in \Sigma^*$ witnessing reachability $c \xrightarrow{w} c'$ should also be provided when the answer is positive.

Example 1. Consider the counter machine given in Figure 2a. This machine is a translation of our running example where the `correct` function simply performs a nonzero test on its argument. All functions are inlined. The location F corresponds to the call to the `critical` function. The `trap` function is modeled in location G by two loops that are chosen non-deterministically (non-determinism typically comes in practice from inputs to the program).

Formally, this counter machine operates on two *counters*, namely x and y, that range over $\mathbb{Z}$. Its *locations* are $\mathtt{A}, \mathtt{B}, \ldots, \mathtt{G}$ and its *edges* are the arrows depicted in Figure 2a. Each edge is labeled with an *action* over the counters. These actions are either guards or assignments. Let Σ denote the set of all counter actions appearing in Figure 2a. The semantics $[\![a]\!]$ of an action $a \in \Sigma$ is defined, as expected, as a binary relation $[\![a]\!] \subseteq \mathbb{Z}^{\{x,y\}} \times \mathbb{Z}^{\{x,y\}}$ over valuations of the counters. The operational semantics of the counter machine is given by the labeled transition system $\mathcal{S} = (C, \Sigma, \to, I, F)$ defined as follows. The set of configurations C is the set of pairs (ℓ, v) where ℓ is a location and $v \in \mathbb{Z}^{\{x,y\}}$ is a valuation of the counters. The sets of initial and final configurations are $I = \mathtt{A} \times \mathbb{Z}^{\{x,y\}}$ and $F = \mathtt{F} \times \mathbb{Z}^{\{x,y\}}$. The labeled transition relation is the set of triples $(\ell, v) \xrightarrow{a} (\ell', v')$ such that $\ell \xrightarrow{a} \ell'$ is an edge depicted in Figure 2a and $(v, v') \in [\![a]\!]$. Our goal can now be formally phrased as the reachability question for $\mathcal{S}$. ◁

We present an algorithm for the reachability problem that is based on symbolic execution. Some additional notations are needed first. A *region* in a LTS $\mathcal{S} = (C, \Sigma, \to, I, F)$ is a subset $\varphi \subseteq C$ of configurations. Regions are often called symbolic states in the context of symbolic execution. We define the region transformer $\mathsf{post} : 2^C \times \Sigma \to 2^C$ as usual, by $\mathsf{post}(\varphi, a) = \{c' \in C \mid \exists c \in \varphi : c \xrightarrow{a} c'\}$.

Symbolic Execution for Reachability Analysis. Symbolic execution has originally been proposed for program testing [14], but the technique can also be used for reachability analysis. Our main contribution concerns exploration strategies for symbolic execution. In order to present and compare these strategies, we first recall some elements about symbolic execution.

An algorithm for reachability analysis based on symbolic execution is given in Algorithm 1. This algorithm takes as input a labeled transition system $\mathcal{S} = (C, \Sigma, \rightarrow, I, F)$ and computes a symbolic reachability tree where each node is labeled with a region (i.e., a subset of C). The set of unprocessed nodes, called the *worklist*, is maintained in the variable W. Initially, the algorithm creates the root of the tree, labeled with the set I of initial configurations, and puts it in the worklist. Then, as long as the worklist is non-empty, the algorithm selects a node from the worklist (more details are given below) and processes it. If the node's region intersects the set F of final configurations then there exists a run from an initial configuration to a final configuration, so the answer "Reachable" is returned. Note that a witnessing trace w can be obtained by collecting the actions along the branch (from the root to the node). Otherwise, the node is expanded, meaning that for each action $a \in \Sigma$, a child is created and labeled with the appropriate region according to the post transformer. This expansion is omitted if the node's region is empty. If the worklist becomes empty then all configurations reachable from an initial configuration have been explored, and none of them is final, so the algorithm returns "Unreachable".

Algorithm 1 `SymbolicExecution`$(\mathcal{S}, \texttt{Prio})$

Input: A LTS $\mathcal{S} = (C, \Sigma, \rightarrow, I, F)$, a priority function $\texttt{Prio} : (\cdots) \rightarrow \mathbb{R} \cup \{+\infty\}$
Output: Either "Reachable" or "Unreachable"

1: $r \leftarrow \texttt{createRoot()}$
2: $(r.\texttt{region}, r.\texttt{priority}) \leftarrow (I, \texttt{Prio}(\mathcal{S}, r, \emptyset))$
3: $W \leftarrow \{r\}$
4: **while** $W \neq \emptyset$ **do**
5: $\quad n \leftarrow \arg\min\{n.\texttt{priority} \mid n \in W\}$
6: $\quad W \leftarrow W \setminus \{n\}$
7: $\quad \varphi \leftarrow n.\texttt{region}$
8: $\quad$ **if** $\varphi \cap F \neq \emptyset$ **then**
9: $\quad\quad$ **return** "Reachable" ▷ *the branch provides a witnessing trace*
10: $\quad$ **else if** $\varphi \neq \emptyset$ **then**
11: $\quad\quad$ **for all** $a \in \Sigma$ **do**
12: $\quad\quad\quad u \leftarrow \texttt{createChild}(n, a)$
13: $\quad\quad\quad (u.\texttt{region}, u.\texttt{priority}) \leftarrow (\mathsf{post}(\varphi, a), \texttt{Prio}(\mathcal{S}, u, W))$
14: $\quad\quad\quad W \leftarrow W \cup \{u\}$
15: $\quad\quad$ **end for**
16: $\quad$ **end if**
17: **end while**
18: **return** "Unreachable"

Remark 1. Algorithm 1 is correct in the sense that it either returns the correct answer to the reachability problem for the input LTS $\mathcal{S} = (C, \Sigma, \rightarrow, I, F)$, or loops forever. The proof is pretty standard. Let $\mathsf{post}^* : 2^C \rightarrow 2^C$ be defined as usual, by $\mathsf{post}^*(\varphi) = \{c' \in C \mid \exists c \in \varphi : c \xrightarrow{*} c'\}$. We introduce in Algorithm 1 a "ghost" variable N that maintains the set of constructed nodes. The correctness of the algorithm follows from the two following properties at line 4. First, $n.\texttt{region}$ is disjoint from F for every node $n \in (N \setminus W)$. Second, $\mathsf{post}^*(I)$ is the union of the set $\bigcup_{n \in (N \setminus W)} n.\texttt{region}$ and the set $\bigcup_{n \in W} \mathsf{post}^*(n.\texttt{region})$. These two properties are routinely shown to be loop invariants at line 4.

In practice, symbolic execution implicitly assumes a maximum exploration depth. The potentially infinite symbolic reachability tree computed by Algorithm 1 is truncated at this maximum exploration depth (and the answer "Unreachable" is replaced by "Unknown" if the tree was truncated).

The order of exploration in Algorithm 1 can be customized via the priority function `Prio`. This function takes three arguments, a LTS, a node and a worklist, and returns a priority in $\mathbb{R} \cup \{+\infty\}$. Each node is assigned a priority upon creation (lines 2 and 13) and this priority remains unchanged afterwards. When the algorithm picks an unprocessed node from the worklist, it picks one of minimal priority (see line 5).

Naturally, the classical search exploration strategies DFS, BFS and NURS can be encoded as priorities. The corresponding priority functions are given by:

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

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

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

The depth-first (DFS) strategy is classically implemented with a last-in-first-out worklist. This strategy is encoded with priorities by ensuring that the last node added to the worklist receives a smaller priority than all other nodes in the worklist (see the `PrioDFS` function). Similarly, the breadth-first (BFS) strategy, which is classically implemented with a first-in-first-out worklist, is equivalent to using the `PrioBFS` function in Algorithm 1. Finally, the `PrioNURS` function provides a random priority for every node added to the worklist, which does correspond to a non-uniform random (NURS) exploration of the tree.

Remark 2. To implement Algorithm 1 in practice, regions have to be finitely representable, emptiness of a region and emptiness of the intersection of two regions have to be decidable, and the post transformer must be computable. In practice, regions are often encoded as SMT formulas.

Remark 3. As in classical symbolic execution, Algorithm 1 blindly expands a node regardless of whether its region has already been processed before. A

computationally cheap inclusion test (i.e., a relation $\preceq$ on regions such that $r \preceq r' \implies r \subseteq r'$) could be used to partially truncate the exploration.

Exploration Strategy Inspired from A*. In addition to the classical strategies DFS, BFS and NURS, we provide a new exploration strategy for symbolic execution, inspired from the A* algorithm.

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. Let us see how to adapt this exploration strategy in our symbolic execution algorithm. In our context, edges are not weighted (they correspond to symbolic transitions $\varphi \xrightarrow{a} \varphi'$ where $\varphi' = \mathsf{post}(\varphi, a)$), so we assume a uniform weight of one. We first need to extend the notion of distance underapproximation to regions.

Definition 1. *A* distance underapproximation *for a LTS $\mathcal{S} = (C, \Sigma, \to, I, F)$ is a function $h_{\mathcal{S}} : 2^C \to \mathbb{N} \cup \{+\infty\}$ such that for every $i, c, f \in C$ and $w \in \Sigma^*$,*

$$i \in I \ \wedge \ i \xrightarrow{*} c \ \wedge \ c \in \varphi \ \wedge \ c \xrightarrow{w} f \ \wedge \ f \in F \quad \implies \quad h_{\mathcal{S}}(\varphi) \leq |w|$$

Informally, $h_{\mathcal{S}}(\varphi)$ returns an underapproximation of the distance between a given region $\varphi \subseteq C$ and the set of final configurations F. However, to facilitate the design of distance underapproximations, this condition on $h_{\mathcal{S}}(\varphi)$ is only required for the configurations $c \in \varphi$ that are reachable from an initial configuration.

To adapt the exploration strategy of A* in Algorithm 1, we assume that we are given a (computable) distance underapproximation $h_{\mathcal{S}}$ for the LTS $\mathcal{S}$ under analysis, and we use the priority function `PrioASTAR` defined as follows:

$$\texttt{PrioASTAR}(\mathcal{S}, u, W, h_{\mathcal{S}}) \ = \ \mathrm{depth}(u) + h_{\mathcal{S}}(u.\texttt{region})$$

where $\mathrm{depth}(u)$ denotes the depth of the node u in the symbolic reachability tree that is generated by Algorithm 1. Note that this is slightly different from A* since $\mathrm{depth}(u)$ only upper-bounds[4] the distance seen so far from the set of initial configurations to the region of u. This is not an issue as our primary goal is to quickly find an executable path, regardless of its length.

Example 2. We illustrate this approach on our running example (see Example 1) by applying it on the LTS giving the semantics of the counter machine

[4] To faithfully mimic A*, $\mathrm{depth}(u)$ should be compared with the depths of all processed nodes having the same region as u. But this would require checking equality between regions, which is computationally costly in general.

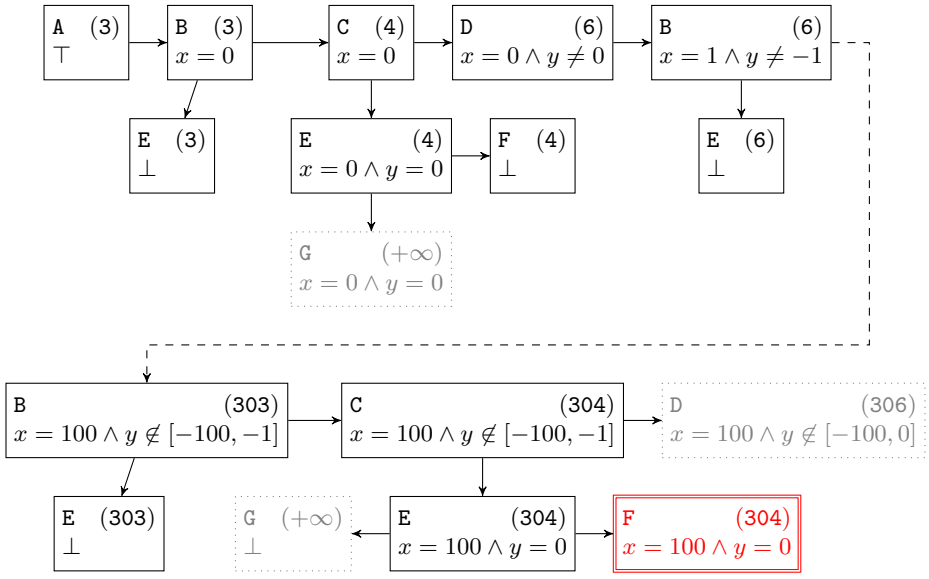


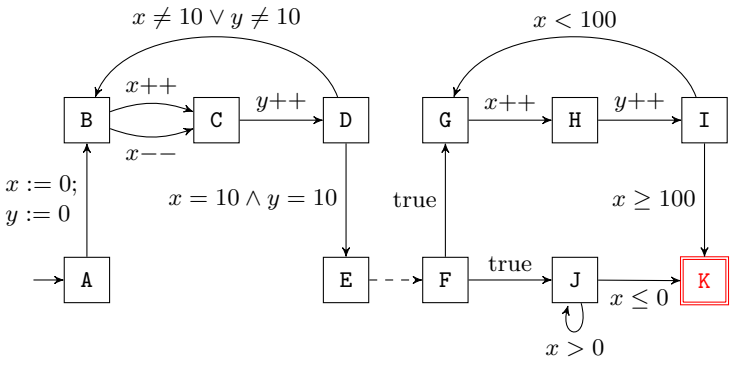
Fig. 3: Symbolic execution with `PrioASTAR` of the counter machine in Figure 2.

of Figure 2a. The symbolic reachability tree generated by Algorithm 1 with the `PrioASTAR` function is (partially) depicted in Figure 3. We use the distance underapproximation obtained by ignoring the counters, given in Figure 2b. Each node in Figure 3 is labeled with its region and its priority (in parentheses). The region is given by a location of the counter machine and a formula over its counters x, y. Recall that the priority of a node is the sum of its depth and of the h value of its location (given in Figure 2b). The order of exploration is not explicitly shown but dotted/gray nodes have not yet been explored and are still in the worklist at the end of the exploration. Our approach explores about 600 nodes before reaching the final location.

In comparison, with a maximum exploration depth of 10000 nodes, at least 10^{270} nodes are explored with `PrioDFS`, assuming that actions are always taken in the same order at line 11 of Algorithm 1. About 10^{30} nodes are explored with `PrioBFS`, most of them stuck in location `G` (this location corresponds to the `trap` function). At least 10^{100} nodes are explored on average with `PrioNURS`. ◁

4 Guiding the Exploration Towards the Unknown

This section presents an improvement of the A*-like exploration strategy presented in the previous section. We first exhibit some weaknesses of this exploration strategy and we then show how to tackle these weaknesses. In short, our improved A*-like exploration strategy steers the exploration towards least explored parts of the system under analysis.



loc.	h(loc.)
A	27
B	26
C	25
D	24
E	23
F	2
G	3
H	2
I	1
J	1
K	0

Fig. 4: Counter machine that illustrates some limitations of our basic A*-like exploration strategy induced by the priority function `PrioASTAR`. The distance underapproximation is shown on the right-hand side.

Limitations of our Basic A*-like Exploration Strategy. In the symbolic reachability tree generated by Algorithm 1 with `PrioASTAR`, the priority of a node u is the sum $\text{depth}(u) + h_{\mathcal{S}}(u.\texttt{region})$. When the non-infinite $h_{\mathcal{S}}$ values are small compared to the depth of the nodes, the resulting exploration roughly amounts to a breadth-first (BFS) exploration (except that nodes u with $h_{\mathcal{S}}(u.\texttt{region}) = +\infty$ are explored last). This is bad news as symbolic execution with BFS is known to perform poorly in practice. Let us illustrate this issue with a small example inspired by our experimentations on Wookey's bootloader (see Section 6).

Example 3. Consider the counter machine given in Figure 4. The two edges $\texttt{B} \xrightarrow{x++} \texttt{C}$ and $\texttt{B} \xrightarrow{x--} \texttt{C}$ model a non-deterministic choice from the location `B`. Similarly, the two edges originating from `F` are chosen non-deterministically. The dashed edge from `E` to `F` stands for 20 intermediate locations between `E` and `F`. This is reflected in the distance underapproximation values given in the table on the right hand-side of the figure. As before, this distance underapproximation is obtained by simply ignoring the counters.

The only run reaching the final location `K` takes the loop `B`–`C`–`D` exactly 10 times, each time choosing the $\texttt{B} \xrightarrow{x++} \texttt{C}$ edge so that x and y remain equal, and exits the loop in `E` with $x = y = 10$. It then moves to `F` and takes the edge $\texttt{F} \xrightarrow{\text{true}} \texttt{G}$ since `K` is not reachable from `J` with $x = 10$. Finally, the loop `G`–`H`–`I` is taken exactly 90 times before moving to `K`.

Symbolic execution with `PrioASTAR` first constructs all nodes obtained by taking the loop `B`–`C`–`D` exactly 9 times, so we end up with 2^9 copies of `B` in the worklist, each with the same depth $1 + 9 \cdot 3$, hence, the same priority 54. Then, the loop `B`–`C`–`D` is taken once more, and exactly one branch exits the loop. This branch reaches `F`, forks into `G` and `J`, and takes the loop on `J` twice. At this point, the worklist contains $2^{10} - 1$ copies of `B` with priority 57, one copy of `G` with priority 56, and one copy of `J` with priority 56. In order to reach the final

location K, the exploration now needs to iterate the loop G–H–I exactly 90 times. But for each iteration of this loop, an additional iteration of the B–C–D loop is performed from each copy of B in the worklist, leading each time to twice as many copies of B in the worklist. This dramatically slows down the construction of the only branch leading to the final location K. ◁

An Improved A*-like Exploration Strategy. As mentioned previously, the issue at hand arises when the sum $\text{depth}(u) + h_{\mathcal{S}}(u.\texttt{region})$ is dominated by $\text{depth}(u)$, which is very common in real-size programs. To fix this issue, we propose to replace $\text{depth}(u)$ 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.

Remark 4. A tempting solution to the above-mentioned issue may be to simply replace $\text{depth}(u)$ by zero, i.e., to let the priority of each node u be $h_{\mathcal{S}}(u.\texttt{region})$. The resulting symbolic execution of the counter machine given in Figure 4 is similar, at first, to the one detailed in Example 3. However, when the branch reaching F forks into G and J, the copy of G now has priority 3 and the copy of J now has priority 1. So the copy of G remains in the worklist and the loop on J is taken forever (or until the maximum exploration depth is reached).

Let us now define the priority function PrioASTAR-2 inducing our improved A*-like exploration strategy. We first introduce the notion that we use to identify "parts of the system". An *observable* for a LTS $\mathcal{S} = (C, \Sigma, \to, I, F)$ is any subset of C. Given a finite set P of observables, we define the region observation function $\mathsf{obs} : 2^C \to 2^P$ by $\mathsf{obs}(\varphi) = \{p \in P \mid (\varphi \cap p) \neq \emptyset\}$. Given a sequence of regions $r_0, \ldots, r_n$, we let $\mathsf{obs}_{\neg\emptyset}(r_0, \ldots, r_n)$ denote the sequence obtained from $\mathsf{obs}(r_0), \ldots, \mathsf{obs}(r_n)$ by removing all occurrences of $\emptyset$.

Observables will be used to focus the exploration on specific properties of the system under analysis. On a given branch of the symbolic reachability tree, instead of looking at the sequence of regions $r_0, \ldots, r_n$ that have been visited along the branch, we will look at the sequence of observations $\mathsf{obs}_{\neg\emptyset}(r_0, \ldots, r_n)$. Typically, for counter machines and binary programs (see Section 5), we consider observables induced by specific locations. But we could use observables expressing properties on counters or registers.

Example 4. In the counter machine of Figure 4, we focus on locations that are targets of branching instructions, i.e., locations in the set $T = \{\texttt{B}, \texttt{C}, \texttt{E}, \texttt{G}, \texttt{J}, \texttt{K}\}$. For each location $t \in T$, we define the observable $p_t = \{t\} \times \mathbb{Z}^{\{x,y\}}$. ◁

The PrioASTAR-2 function is defined in Algorithm 2. To simplify the presentation, we assume that the set I of initial configurations has a nonempty observation. This guarantees that the sequence $obs_0, \ldots, obs_k$ defined at line 2 is nonempty. The priority returned by PrioASTAR-2 is $g \cdot \lambda(\mu) + h_{\mathcal{S}}(u.\texttt{region})$ where g and μ depend on the sequence $obs_0, \ldots, obs_k$ of nonempty observations seen along the branch. In words, g is the "elementary" length of this sequence,

Algorithm 2 `PrioASTAR-2`$(\mathcal{S}, u, W, h_{\mathcal{S}}, P, \lambda)$

Input: A LTS $\mathcal{S} = (C, \Sigma, \to, I, F)$, a node u, a worklist, a distance underapproximation $h_{\mathcal{S}}$ for $\mathcal{S}$, a finite set P of observables for $\mathcal{S}$, a function $\lambda : \mathbb{N} \to \mathbb{R}_{\geq 0}$

1: Let $u_0, \ldots, u_n$ denote the branch from the root $r = u_0$ to the node $u = u_n$
2: Let $obs_0, \ldots, obs_k = \mathtt{obs}_{\neg\emptyset}(u_0.\mathtt{region}, \ldots, u_n.\mathtt{region})$ ▷ $k \geq 0$
3: Let $g = \mathrm{Card}\{obs_0, \ldots, obs_m\}$ where $m = \min\{i \in [0, k] \mid obs_i = obs_k\}$
4: Let $\mu = \mathrm{Card}\{i \in [0, n] \mid obs_i = obs_k\}$
5: **return** $g \cdot \lambda(\mu) + h_{\mathcal{S}}(u.\mathtt{region})$

i.e., the number of distinct elements in the sequence $obs_0, \ldots, obs_m$ where obs_m is the first occurrence of obs_k, and μ is the number of times that obs_k occurs in the sequence $obs_0, \ldots, obs_k$. Intuitively, obs_k indicates which part of the system corresponds to the node u, so μ tells us how many times this part of the system has been visited along the branch. Observe that g only depends on the first occurrence of each observation in $obs_0, \ldots, obs_k$. We call g the *elementary depth* of the node u.

The function $\lambda : \mathbb{N} \to \mathbb{R}_{\geq 0}$ allows us to adjust the priority depending on the value of μ. The choice of a good λ function is crucial to guide the exploration properly. In order to steer the exploration towards least explored parts of the system, λ should be non-decreasing, and $\lambda(\mu)$ should be small when μ is small. According to our experiments, a λ function of the form

$$\lambda_\theta(\mu) \;=\; \begin{cases} 0 & \text{if } \mu < \theta \\ \log_{10}(\mu - \theta + 1) & \text{otherwise} \end{cases}$$

performs well in practice. Here, the parameter $\theta \in \mathbb{N}$ acts as a threshold (in our experiments, we use $\theta = 3$, see Section 6). The idea behind λ_θ is to give precedence to nodes that are in a part of the system that has rarely been visited (less than θ times) along the branch. Note that this function always returns zero or a small value. As mentioned before, we do this to prevent the elementary depth g from dominating $h_{\mathcal{S}}$. Note that this is just an example of a possible λ function that we designed during our experimentations. Different λ functions may also work, and even outperform this one.

Example 5. Consider again the counter machine given in Figure 4. We take the same set of observables as in Example 4, and we use the function λ_θ defined above with $\theta = 3$. As with `PrioASTAR`, symbolic execution with `PrioASTAR-2` first constructs all nodes obtained by taking the loop `B`–`C`–`D` exactly 9 times. When the branch that exits the loop forks into `G` and `J`, the worklist contains $2^{10} - 1$ copies of `B` with priority $26 + 1 \cdot \log_{10}(8)$, one copy of `G` with priority 3, and one copy of `J` with priority 1. So the loop on `J` is iterated first, and the priority of the `J` copy in the worklist slowly increases. When this priority becomes larger than 3, the loop `G`–`H`–`I` is also iterated. The exploration then interleaves the construction of the two corresponding branches. After 90 iterations of the loop `G`–`H`–`I`, the worklist contains a copy of `K` with priority 0. This copy is then processed

immediately, and the algorithm returns "Reachable". Let us estimate the number of iterations of the loop on `J`. Just before completion of the `G`–`H`–`I` loop, the last copy of `G` in the worklist has priority $3+4\cdot\log_{10}(90-1) < 3+4\cdot 2 = 11$. Similarly, the last copies of `H` and `I` have priorities less than 8 and 7, respectively. After $k \geq 3$ iterations of the loop on `J`, the priority of the `J` copy is $1+4\cdot\log_{10}(k-1)$. Observe that $(1+4\cdot\log_{10}(k-1) > 11) \Leftrightarrow k > 317$. So the loop on `J` is iterated at most 318 times. Note also that the $2^{10}-1$ copies of `B` have not left the worklist since their priority is larger than 26, hence, larger than 11. ◁

5 Application to Binary Programs

We show in this section how to apply our approach to binary programs. Recall that our new A*-like exploration strategies require a distance underapproximation for the LTS under analysis. The main purpose of this section is to provide an efficiently computable distance underapproximation for binary programs. Before that, we need to define[5] the syntax and semantics of binary programs.

Syntax and Semantics. Consider a fixed set *Reg* of *registers* and a fixed set *Addr* of *addresses*. To account for instructions that do not impact the control-flow of the program, such as memory accesses and arithmetic operations on registers, we assume an a priori given set `Op` of *operations*. Each operation $\texttt{op} \in \texttt{Op}$ comes with its semantics $[\![\texttt{op}]\!]$, given as a function from $\mathbb{Z}^{Reg} \times \mathbb{Z}^{Addr}$ to itself. A *binary program* is a finite sequence of instructions $(I_1, \ldots, I_n)$, where each *instruction* I_k is in the following set:

$$\texttt{Op} \;\cup\; \{\texttt{BR}\ r\ \ell \mid r \in Reg \wedge \ell \in [1,n]\} \;\cup\; \{\texttt{CALL}\ \ell \mid \ell \in [1,n]\} \;\cup\; \{\texttt{RET}\}$$

Here, `BR` stands for conditional branching, and `CALL` and `RET` stand for procedures call and return. A *location* of the binary program is any integer in $[1, n+1]$.

The operational semantics of a binary program $(I_1, \ldots, I_n)$, equipped with a final location $f \in [1, n+1]$, is given by the labeled transition system $\mathcal{S} = (C, \Sigma, \rightarrow, I, F)$ defined as follows. The set of actions Σ is the set of instructions of the programs, i.e., $\Sigma = \{I_1, \ldots, I_n\}$. The set of configurations C is the set of quadruples (ℓ, R, M, s) where $\ell \in [1, n+1]$ is a location, $R \in \mathbb{Z}^{Reg}$ and $M \in \mathbb{Z}^{Addr}$ are register and memory contents, and $s \in [1, n+1]^*$ is a stack contents. The sets of initial and final configurations are $I = \{(\ell, R, M, s) \in C \mid \ell = 1 \wedge s = \varepsilon\}$ and $F = \{(\ell, R, M, s) \in C \mid \ell = f\}$. The labeled transition relation $\rightarrow$ is defined by the rules given in Figure 5. Note that each of these rules implicitly requires that $\ell \in [1, n]$ since I_ℓ must be defined.

Distance Underapproximation. Following the approach of Blondin et al. for Petri nets [4], we propose a distance underapproximation for binary programs

[5] Similar definitions of the syntax and semantics of binary programs can be found in the literature. Our definition is intentionally simple and tailored to our purposes.

$$\frac{I_\ell = \texttt{op} \in \texttt{Op} \qquad (R', M') = [\![\texttt{op}]\!](R, M)}{(\ell, R, M, s) \xrightarrow{I_\ell} (\ell + 1, R', M', s)}$$

$$\frac{I_\ell = \texttt{BR}\ r\ \ell' \qquad R(r) = 0}{(\ell, R, M, s) \xrightarrow{I_\ell} (\ell + 1, R, M, s)} \qquad \frac{I_\ell = \texttt{BR}\ r\ \ell' \qquad R(r) \neq 0}{(\ell, R, M, s) \xrightarrow{I_\ell} (\ell', R, M, s)}$$

$$\frac{I_\ell = \texttt{CALL}\ \ell'}{(\ell, R, M, s) \xrightarrow{I_\ell} (\ell', R, M, (\ell + 1) \cdot s)} \qquad \frac{I_\ell = \texttt{RET}}{(\ell, R, M, \ell' \cdot s) \xrightarrow{I_\ell} (\ell', R, M, s)}$$

Fig. 5: Operational semantics of binary programs.

$$\frac{I_\ell = \texttt{op} \in \texttt{Op}}{(\ell, s) \xrightarrow{I_\ell}{}^\sharp (\ell + 1, s)} \qquad \frac{I_\ell = \texttt{BR}\ r\ \ell'}{(\ell, s) \xrightarrow{I_\ell}{}^\sharp (\ell + 1, s)} \qquad \frac{I_\ell = \texttt{BR}\ r\ \ell'}{(\ell, s) \xrightarrow{I_\ell}{}^\sharp (\ell', s)}$$

$$\frac{I_\ell = \texttt{CALL}\ \ell'}{(\ell, s) \xrightarrow{I_\ell}{}^\sharp (\ell', (\ell + 1) \cdot s)} \qquad \frac{I_\ell = \texttt{RET}}{(\ell, \ell' \cdot s) \xrightarrow{I_\ell}{}^\sharp (\ell', s)}$$

Fig. 6: Abstract semantics of binary programs.

that is based on an abstraction of the operational semantics defined above. This abstraction merely consists of ignoring the register and memory contents.

Formally, the abstract semantics of a binary program $(I_1, \ldots, I_n)$, equipped with a final location $f \in [1, n + 1]$, is given by the labeled transition system $\mathcal{S}^\sharp = (C^\sharp, \Sigma, \rightarrow^\sharp, I^\sharp, F^\sharp)$ defined as follows. The set of actions Σ is the same as before, i.e., $\Sigma = \{I_1, \ldots, I_n\}$. The set of abstract configurations $C^\sharp$ is the set of pairs (ℓ, s) where $\ell \in [1, n + 1]$ is a location and $s \in [1, n + 1]^*$ is a stack contents. The sets of initial and final abstract configurations are $I^\sharp = \{(1, \varepsilon)\}$ and $F^\sharp = \{f\} \times [1, n+1]^*$. The labeled abstract transition relation $\rightarrow^\sharp$ is defined by the rules given in Figure 6. Again, each of these rules implicitly requires that $\ell \in [1, n]$. Obviously, every run in $\mathcal{S}$ can be mimicked in $\mathcal{S}^\sharp$ by ignoring the register and memory contents. Formally, it holds that $(\ell, s) \xrightarrow{w}{}^\sharp (\ell', s')$ in $\mathcal{S}^\sharp$ when $(\ell, R, M, s) \xrightarrow{w} (\ell', R', M', s')$ in $\mathcal{S}$. So we can use $\mathcal{S}^\sharp$ to underestimate the distance in $\mathcal{S}$ between two (sets of) configurations.

For efficiency reasons, our distance underapproximation is based on the precomputation of the distance in $\mathcal{S}^\sharp$ between pairs of locations $\ell, \ell' \in [1, n + 1]$. However, if we start in ℓ with an arbitrary stack contents, then a RET instruction may directly lead to ℓ'. This would yield an extremely coarse distance underapproximation. So we restrict the stack contents to "legitimate" ones, in the sense that the stack starts with a valid return location. Formally, we say that an abstract configuration (ℓ, s) is *coherent* if s is empty or of the form $s = \ell' \cdot s'$ with $\ell' \in [1, n+1]$ such that $(\ell'', \varepsilon) \xrightarrow{*}{}^\sharp (\ell, \ell')$ for some $\ell'' \in [1, n+1]$. Note that every

abstract configuration (ℓ, s) reachable in $\mathcal{S}^\sharp$ from $(1, \varepsilon)$ is coherent. A run in $\mathcal{S}^\sharp$ is called *coherent* when all abstract configurations visited by the run (including the first and last ones) are coherent. We write $(\ell, s) \xrightarrow{w}{}^\sharp_{\text{co}} (\ell', s')$ when there exists a coherent run from (ℓ, s) to (ℓ', s') with trace w. Let $d^\sharp : [1, n+1] \times [1, n+1] \to \mathbb{N}$ be defined[6] by:

$$d^\sharp(\ell, \ell') \;=\; \inf\{|w| \mid \exists s, s' \in [1, n+1]^* : (\ell, s) \xrightarrow{w}{}^\sharp_{\text{co}} (\ell', s')\}$$

The distance underapproximation $h_{\mathcal{S}}$ that we propose is defined as follows:

$$h_{\mathcal{S}}(\varphi) \;=\; \inf\{d^\sharp(\ell, f) \mid (\ell, R, M, s) \in \varphi\}$$

Intuitively, $h_{\mathcal{S}}(\varphi)$ is the minimal distance from the locations of φ to f in the abstract semantics $\mathcal{S}^\sharp$ restricted to coherent abstract configurations. It is readily seen that the function $h_{\mathcal{S}}$ satisfies the condition of Definition 1.

Remark 5. Our notion of coherence for abstract configurations only accounts for the top-most return location on the stack. The remainder of the stack may be arbitrary. However, for every coherent run $(\ell, \ell_1 \cdots \ell_k \cdot s) \xrightarrow{*}{}^\sharp_{\text{co}} (\ell', s)$, the prefix $\ell_1 \cdots \ell_k$ of the stack that is popped in the run is "legitimate" in the sense that each ℓ_i is a valid return location for ℓ_{i-1} (formally, the abstract configurations (ℓ_{i-1}, ℓ_i) are coherent).

To compute $h_{\mathcal{S}}(\varphi)$, we need to compute $d^\sharp(\ell, f)$ for every location ℓ. First, we compute, for each instruction `CALL` ℓ appearing in the binary program, the ℓ-summary $\inf\{|w| \mid \exists \ell' : (\ell, \varepsilon) \xrightarrow{w}{}^\sharp (\ell', \varepsilon) \wedge I_{\ell'} = \texttt{RET}\}$. Second, we compute the values $d^\sharp(\ell, f)$ by applying a single-source shortest path algorithm on $\mathcal{S}^\sharp$ augmented with summaries, starting from f and moving backwards on edges. The resulting algorithm is similar to the one described in [2].

Wrap Up. We now have the necessary ingredients to perform symbolic execution (see Algorithm 1) with our new A*-like exploration strategies. Regions use SMT formulas for register and memory contents, and an explicit representation for locations and stack contents. The `post` transformer is computed by following the operational semantics given in Figure 6. The distance underapproximation provided to `PrioASTAR` and `PrioASTAR-2` is the one presented in the previous subsection. Finally, the finite set P of observables given to `PrioASTAR-2` is induced by the locations that are targets of control-flow instructions. Let T denote these locations, i.e., T is the set of all $t \in [1, n+1]$ such that there exists $\ell, \ell' \in [1, n]$ and $r \in \mathit{Reg}$ verifying $I_\ell \in \{\texttt{BR}\ r\ \ell', \texttt{CALL}\ \ell'\}$ and $t \in \{\ell', \ell + 1\}$. Formally, P is the set of all subsets $p_t = \{t\} \times \mathbb{Z}^{\mathit{Reg}} \times \mathbb{Z}^{\mathit{Addr}} \times [1, n+1]^*$ where t ranges over T.

6 Experimental results

We evaluate our new approach on seven programs: the two running examples of the paper (Figures 2 and 4), three "*crackme*" challenges [20–22] which are rel-

[6] Recall that $\inf X = \min X$ for every non-empty subset $X \subseteq \mathbb{N}$ and that $\inf \emptyset = +\infty$.

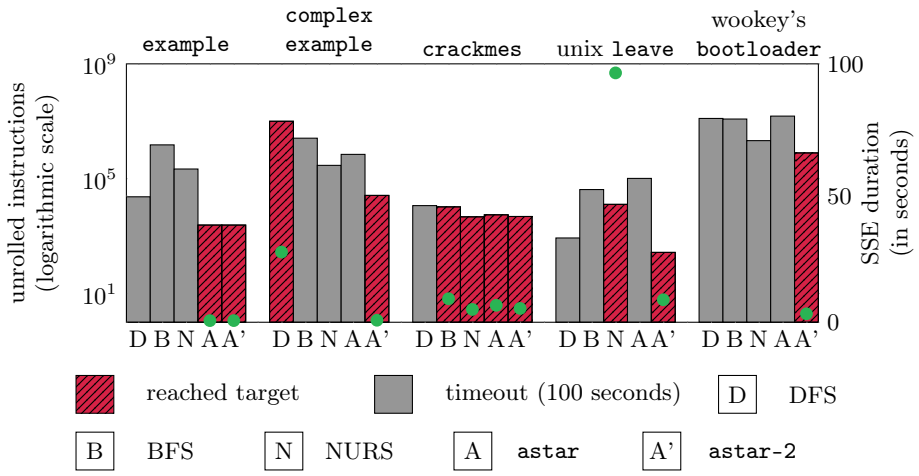


Fig. 7: Experimental results obtained with Binsec: bars represent the number of unrolled instructions in a logarithmic scale and dots represent the SSE duration in seconds.

atively 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[7] meant to be robust against various type of attacks (∼10K locations), and the `leave` command of NetBSD (∼100K locations). The programs are cross-compiled to pure THUMB-2 with target CPU *cortex-m3* and *armv7-m* architectures.

We use the symbolic execution tool Binsec (version 0.6), in which we have implemented our new strategies. The targets for the study are chosen arbitrarily but meant to be deep in the execution flow or difficult to reach. A time limit of 100 seconds is allowed for each experiment, beyond which we stop it and report a timeout (*t.o.*). In the Wookey bootloader and the `leave` programs, we have stubbed some parts of the code to accelerate the process. All benchmarks were ran on a AMD64 Oracle Linux Server (release 8.8) machine with an Intel(R) Xeon(R) Gold 6244 CPU (3.60 GHz) and with 256GiB of RAM. A replication package for our experiments (including the source code and the seven programs) is available at Zenodo [10].

We compare the approach based on `PrioASTAR-2` (named `astar-2` in the benchmarks) with the usual exploration strategies (`dfs`, `bfs` and `nurs`), and also with our basic A*-like approach, i.e., based on `PrioASTAR` (named `astar` in the benchmarks). In Figure 7, we compare the number of unrolled instructions and the symbolic execution time for each exploration strategy on the different programs. The results of the three crackmes are summed up and displayed as one "program" named `crackmes`. In the case of NURS, the experiments are

[7] French National Cybersecurity Agency.

ran 10 times and the average for both metrics are displayed. The exploration strategies are on the X-axis, the number of unrolled instructions is on the left Y-axis and represented by the bars. Finally, the symbolic execution time is on the right Y-axis and displayed by green dots. For readability reasons, we use a logarithmic scale for the number of unrolled instructions. 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.

The efficiency of the exploration depends on the maximum exploration depth. The perfect bound is not definable beforehand so we set it to 10^7 instructions for all programs. Finally, the results of our exploration strategy `astar-2` depend on the function λ (see Section 4). The best λ function is specific to each situation, nevertheless we chose to systematically use λ_θ with $\theta = 3$ in our experiments. Using a smaller parameter θ tends to steer the exploration towards a BFS, while a larger parameter θ steers the exploration towards a DFS. The best in-between value we found was $\theta = 3$.

7 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. Although some faulty execution may still remain difficult to catch, this approach shows promising results. Our key insight while designing this algorithm is to create a balanced mix between a DFS and a BFS. The strategy has been designed on generic transition systems, making it applicable in various situations. We have described how to apply it on binary code, and provided an experimental evaluation showing that our strategy outperforms the classical exploration strategies DFS, BFS and NURS and scales well on real-size programs. As future work, we intend to apply this technique to the detection of hardware vulnerabilities (i.e., vulnerabilities to fault injection attacks).

Acknowledgements. This work was supported by the French ANRT CIFRE 2021/1673 Project. We also would like to thank Guillaume Baud-Berthier, Julien Bernet and Michael Grand for their helpful discussions.

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. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Handbook of satisfiability **185**(99), 457–481 (2009)
4. 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)
5. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
6. Chess, B., McGraw, G.: Static analysis for security. IEEE security & privacy **2**(6), 76–79 (2004)
7. Clarke Jr, E.M., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model checking. MIT press (2018)
8. Cousot, P.: Abstract interpretation. ACM Computing Surveys (CSUR) **28**(2), 324–328 (1996)
9. David, R., Bardin, S., Ta, T.D., Mounier, L., Feist, J., Potet, M.L., Marion, J.Y.: Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). vol. 1, pp. 653–656. IEEE (2016)
10. De Castro Pinto, T., Rollet, A., Sutre, G., Tobor, I.: Replication package for "Guiding Symbolic Execution with A-star" (2023). https://doi.org/10.5281/zenodo.8169445
11. 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)
12. Ducousso, S., Bardin, S., Potet, M.L.: Adversarial reachability for program-level security analysis. Programming Languages and Systems LNCS 13990 p. 59 (2023)
13. 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)
14. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976)
15. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. Formal aspects of computing **27**(3), 573–609 (2015)
16. Li, J., Zhao, B., Zhang, C.: Fuzzing: a survey. Cybersecurity **1**(1), 1–13 (2018)
17. Li, Y., Su, Z., Wang, L., Li, X.: Steering symbolic execution to less traveled paths. ACM SigPlan Notices **48**(10), 19–32 (2013)
18. 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)

19. MIASM: Cea-sec. https://github.com/cea-sec/miasm (2015)
20. NoraCodes: crackmes. https://github.com/NoraCodes/crackmes/blob/master/crackme03.c (2017)
21. NoraCodes: crackmes. https://github.com/NoraCodes/crackmes/blob/master/crackme05.c (2017)
22. NoraCodes: crackmes. https://github.com/NoraCodes/crackmes/blob/master/crackme09.c (2017)
23. Potet, M.L., Mounier, L., Puys, M., Dureuil, L.: Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. pp. 213–222. IEEE (2014)
24. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: IEEE Symposium on Security and Privacy (2016)
25. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS. vol. 16, pp. 1–16 (2016)
26. 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)