Guided Dynamic Symbolic Execution Using Subgraph Control-Flow Information

Conference Paper · July 2016

DOI: 10.1007/978-3-319-41591-8_6

3 authors, including:

Josselin Feist
Verimag
14 PUBLICATIONS **146** CITATIONS
SEE PROFILE

Laurent Mounier
University of Grenoble
83 PUBLICATIONS **2,402** CITATIONS
SEE PROFILE

Guided Dynamic Symbolic Execution Using Subgraph Control-Flow Information

Josselin Feist, Laurent Mounier, Marie-Laure Potet

Univ. Grenoble Alpes, F-38000 Grenoble, France

Abstract. Dynamic symbolic execution (DSE) is an efficient SMT-based path enumeration technique used in software testing. In this work in progress, we consider here the case of guided DSE, where the paths to enumerate should be part of a given program slice. We propose a new path selection criterion, which aims to minimize the number of queries to the SMT solvers. This criterion is based on the probability of a path to exit the program slice. Experiments show that this information can be computed in a reasonable time for DSE purpose.

1 Guided Dynamic Symbolic Execution

Dynamic symbolic execution (DSE) is a technique used in software testing and vulnerability analysis. This subject has received a large interest these past years [5]. DSE mixes a concrete execution trace and a symbolic reasoning on it. From a given symbolic execution path, a logical formula called *path predicate* is built, from which conditional instructions can be inverted using an SMT solver. This operation leads to the generation of a new *input*, which can be used to obtain a new path, and so on. Exploring all bounded paths in a software is not realistic, due to the large number of paths. This limitation is well known as the *path explosion* problem [2]. A key feature of DSE is thus the strategy used to select which part of the program should be explored first, either to maximize path coverage or to reach specific locations. Our work falls in the latter category: exploration is led towards a goal and focuses on a specific part of the program. This approach is called Guided Dynamic Symbolic Execution. For example, [8] uses DSE to reach a given instruction, while [1, 6] use it to confirm results coming from static analysis. In order to reach a goal, two kinds of strategies are used: *Control Flow Guided* and *Data Flow Guided*. For example, [8] belongs to the first family: distance between the source and the destination is used to select which part of the program will be explored first. Meanwhile, [6] belongs the second one: taint analysis is used to guide the exploration.

We focus on *Control Flow Guided* strategies. We observed that most of the strategies in this family are not well adapted in a context of program slice exploration. We propose a new metric based on subgraph information, that we combine with random walks to guide the exploration. The paper is organized as follows. In Section 1.1, we give a motivating example and explain limitations of state of the art techniques on it. Then

we present our subgraph representation in Section 2.1 and define our new metric in Section 2.2. We give a criterion on subgraph extraction for which our approach is well adapted in Section 2.3. Afterwards, we present related work on guided strategies in Section 3. Finally, in Section 4 we discuss possible improvements.

1.1 Motivating Example

Figure 1 is used to illustrate our proposition. The right side of the Figure is the control-flow graph representation of the source code, where node numbers fit with source code lines. We assume that the objective of the DSE is to reach the call to function `goal` (line 11), and that `long_computation` (line 9) is a very large subgraph, with all internal paths leading to node 11. The explored slice is represented by nodes in the subgraph: $\{3, 4, 9, 11\}$ (nodes in the dotted square). Starting from node 3, there are three path categories: *(i)* paths that do not reach destination node 11 and so are outside the subgraph (called $Path_{out}$) *(ii)* paths that reach destination node 11 through node 4 (called $Path_{sp}$, those in shortest paths), and *(iii)* paths that reach destination node 11 through node 9 (called $Path_{lp}$, those in longest paths). Paths in $Path_{lp}$ are assumed to be significantly longer[1] than paths in $Path_{sp}$ and $Path_{out}$. Paths containing the node 4 can either be in $Path_{out}$ or $Path_{sp}$. So, going through node 4 can potentially lead to not reach the destination. Conversely, paths containing node 9 can only be in $Path_{lp}$ and choosing such paths will ensure to reach the destination.

```
1  void f()
2  {
3   if(..){
4    if(..){
5     go_out();
6    }
7   }
8   else{
9    long_computation;
10  }
11  goal();
12 }
```

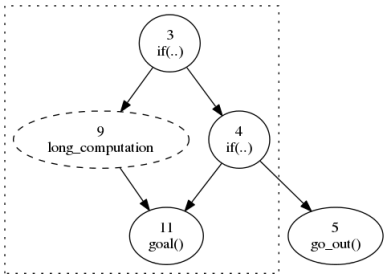

Fig. 1: Motivating example

1.2 Proposition

We observe that most *Control Flow Guided* strategies are based on shortest path, without taking into account the context of program slice exploration (see Section 3). In Figure 1, state of the art strategies will first

[1] In number of nodes and comparisons.

select node 4 rather than node 9 in order to reach node 11 from node 3. However, this choice is clearly not the most appropriate. A path going through node 9 requires no further inversion, while passing through node 4 there is a 50% chance to need an inversion (that requires one more solver query). We notice that classical exploration of program slice does not take into account branches that do not reach the destination. For example, by removing node 5 and its incoming edge, the information needed to choose node 9 over 4 is lost. Our metric is based on this particular information.
We propose a new path selection heuristic, based on the number of instruction inversions necessary to reach the goal, rather than on the length of a path. More specifically, we compute the probability of a path starting from a node to reach the destination, while taking into account that it can go out of the subgraph.

2 Using Subgraph Control-Flow Information

2.1 Subgraph Transformation

We define $G = (V, E)$, where G is the control-flow graph representation of the analyzed program, V a set of nodes n_i and E a set of directed edges. $e_{i,j}$ denotes the edge between nodes n_i and n_j. A program slice is given as a pair (n_{dst}, φ) where $n_{dst} \in V$ and φ is a property on paths. $G_{dst} = (V', E')$ is the subgraph of G with n_{dst} as destination node. V' is the set of nodes satisfying φ. All nodes that do not satisfy φ are merged into one single node: n_{out}. In a context of line reachability, φ states if a node can reach the destination or not. In more complex contexts, as in static analysis validation, φ can describe more specific properties [4]. n_{out} and n_{dst} are both absorbing nodes, meaning that we replace their outedges by self-loops. Figure 2 explains the subgraph extraction algorithm.

$$
\begin{aligned}
V' = & \quad \{v \in V \mid \varphi(v, n_{dst})\} \cup \{n_{out}\} \\
E' = & \{e_{i,j} \in E \mid n_i \in V' \wedge n_j \in V' \wedge n_i \neq n_{dst}\} \cup \\
& \{e_{i,out} \mid \exists e_{i,j} \in E \wedge n_i \in V' \wedge n_j \notin V'\} \cup \\
& \{e_{dst,dst}\} \cup \{e_{out,out}\}
\end{aligned}
$$

Fig. 2: Subgraph transformation

2.2 Using Random Walk to Guide the Exploration

The main idea is now to compute the probability to reach rather n_{dst} than n_{out}. From a node, this probability can be seen as the number of elementary[2] paths reaching n_{dst} rather than n_{out}. Unfortunately, elementary paths computation is exponential. In order to be scalable, we

[2] A elementary path is a path where no node appears more than once.

propose a more realistic heuristic. A way to approximate program paths is to use random walks. Computing the probability for a random walk to reach n_{dst} starting from a node yields the desired probability. This can be computed using the *transition matrix* of the graph [7]. *A transition matrix* T is a $|N|*|N|$ matrix where $|N|$ is the number of nodes and $T(n_i, n_j)$ represents the probability for a random walk to move from n_i to n_j in one step. If there is no edge between n_i and n_j, this probability is 0, otherwise it is equal to 1 divided by the number of out-edges of n_i in a unweighted graph:

$$T(n_i, n_j) = \begin{cases} 0 & \text{if } e_{i,j} \notin E, \\ \frac{1}{deg_{out}(n_i))} & \text{otherwise.} \end{cases}$$

An absorbing node a is a special node, where $T(a, a) = 1$ and $T(a, j) = 0, \forall j \neq a$. Absorbing nodes are used to stop the random walk. From our subgraph transformation (Section 2.1), n_{dst} and n_{out} are both absorbing nodes. Then $T^l(i, j)$ represents the probability of a random walk to be in j starting from i after l steps [7]. An interesting point is that we focus only on two specific destinations: n_{dst} and n_{out}. Since they are absorbing nodes, paths represented by $T^l(n_i, n_{dst})$ (resp. $T^l(n_i, n_{out})$) contain the one represented by $T^{l-1}(n_i, n_{dst})$ (resp. $T^{l-1}(n_i, n_{out})$). For our example in Figure 1 we have[3]:

$$T = \begin{array}{c} \\ n_3 \\ n_4 \\ n_{out} \\ n_9 \\ n_{11} \end{array} \begin{array}{c} \begin{array}{ccccc} n_3 & n_4 & n_{out} & n_9 & n_{11} \end{array} \\ \begin{pmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \qquad T^2 = \begin{array}{c} \\ n_3 \\ n_4 \\ n_{out} \\ n_9 \\ n_{11} \end{array} \begin{array}{c} \begin{array}{ccccc} n_3 & n_4 & n_{out} & n_9 & n_{11} \end{array} \\ \begin{pmatrix} 0 & 0 & \frac{1}{4} & 0 & \frac{3}{4} \\ 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

We define $P^l(n_i, e_{i,j})$ as the probability to reach the destination n_{dst} in at most l steps starting from n_i and going through $e_{i,j}$, as follows:

$$P^l(n_i, e_{i,j}) = T^l(n_j, n_{dst})$$

P^l is computed before the DSE exploration and it is used as score during the exploration to prioritize over the choice of edges. In case of equality, a classical shortest path, or $T^l(n_i, n_{out})$[4], can be used to settle the choice. In our example, $P^2(n_3, e_{3,4}) = T^2(n_4, n_{11}) = \frac{1}{2}$ and $P^2(n_3, e_{3,9}) = T^2(n_9, n_{11}) = 1$, so $e_{3,9}$ is chosen.

2.3 Subgraph Pattern

Our proposed strategy makes sense only if nodes in shortest paths could lead out of the subgraph. Yet this corresponds to concrete programming patterns as shown in Figure 3. The first one is close to the example given in Figure 1. Here, the `true` branch of a comparison leads to a short function, but with paths inside this function that do not satisfy the property φ (see section 2.1). On the contrary, the `false` branch leads to a longer

[3] There are at most 2 steps in this example, so we choose $l = 2$.

[4] $T^l(n_i, n_{dst}) + T^l(n_i, n_{out}) \leq 1$ since a random walk ends not necessary in an absorbing node after l steps.

function, with all paths satisfying φ. The second pattern appears every time there is a list of comparisons and only the first one contains another comparison leading paths to not satisfy φ. More generally, our approach differs from shortest path algorithms whenever subgraph respects the following criterion:

Criterion 1 *In the subgraph, nodes in the shortest paths to n_{dst} also appear in numerous paths leading out of the subgraph.*

```
if(cond)
 small_calc() // small_calc
      contains paths that do
      not satisfy the desired
      property
else
 long_calc() // in long_calc
      all paths satisfy the
      desired property
```

```
if(cond){
 if(cond){
          ...  // the desired
               property is not
               satisfy
 }
 return ;
}
else if(cond){
 return ;
}
return;
```

Fig. 3: Pattern examples

2.4 Overhead

Our approach is not yet implemented in a DSE. However we compute P^l on a slice coming from *Jasper-JPEG-2000*[5]. The slice contains 2000 nodes and 2600 edges. For random walks with a length of 200000 steps ($l = 200000$), the computation takes 8 seconds. Since we only need to compute P one time (as preprocessing), it is clearly negligible compared to the computation time needed by a guided DSE to explore this slice.

3 Related Work

As discussed in previous sections, state of the art strategies are mostly based on different variants of shortest path and on data-flow analysis. For example, [8] mixes a shortest path analysis with a backward analysis to reach a specific line of code. [11] uses a *proximity heuristics* that computes the shortest path on basic blocks. [9] combines shortest paths on conditional instructions with a data-flow analysis to remove unreachable paths. In [1], data-flow analysis is combined with shortest path in the *Visible Pushdown Automaton* (VPA) representation of the program. [12] proposes to join a data-flow analysis with *Finite State Machine* (FSM)

[5] `https://www.ece.uvic.ca/~frodo/jasper/#overview`

to select a path that satisfies a property as soon as possible. There are many other strategies and it is not in the scope of this paper to list them all. Yet, to the best of our knowledge, there are no *Control Flow Guided* strategies that are not based on shortest paths.

4 Conclusion and Perspectives

We present in this paper the use of a new heuristic, using control-flow information and random walks to guide a DSE towards a goal in a program slice. It still has a large possibility of improvement. First of all, our metric needs to be integrated in a DSE and compared with state of the art strategies. It would be also relevant to test our metric on results of static analysis [4, 10], to determine if some analyses create subgraphs that fit well with Criterion 1. We plan to integrate our work inside the BINSEC/SE framework [3] and use it in a security oriented purpose. More specifically, this work is driven by the need to confirm results coming from an *use-after-free* static analyzer [4]. Another perspective is also to correlate our approach with data-flow analysis, by weighing the random walk from its results, or directly during subgraph transformation. We also believe that strategies in DSE exploration currently lack of an adapted use of graph theory metrics. One of future directions is to better integrate these notions in DSE usage.

References

1. Babic, D., Martignoni, L., McCamant, S., Song, D.: Statically-directed dynamic automated test generation. In: ISSTA (2011)
2. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM (2013)
3. David, R., Bardin, S., Ta, T.D., Feist, J., et al: Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. SANER (2016)
4. Feist, J., Mounier, L., Potet, M.L.: Statically detecting Use-after-Free on binary code (JCVHT 2014)
5. Godefroid, P.: 500 machine-years of software model checking and smt solving (invited speaker). SEFM (2014)
6. Haller, I., Slowinska, A., Neugschwandtner, M., Bos, H.: Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In: USENIX SEC (2013)
7. Lovász, L.: Random walks on graphs: A survey. Combinatorics, Paul Erdos is Eighty (1993)
8. Ma, K.K., Khoo, Y.P., Foster, J.S., Hicks, M.: Directed symbolic execution. In: SAS (2011)
9. Marinescu, P.D., Cadar, C.: Katch: high-coverage testing of software patches. In: ESEC/SIGSOFT FSE (2013)
10. Rawat, S., Mounier, L.: Finding buffer overflow inducing loops in binary executables. In: SERE (2012)
11. Zamfir, C., Candea, G.: Execution synthesis: a technique for automated software debugging. In: EuroSys (2010)
12. Zhang, Y., Chen, Z., Wang, J., Dong, W., Liu, Z.: Regular property guided dynamic symbolic execution. In: ICSE (2015)