# Bounded Local Reachability Analysis in Pre-compiled Applications

Horatiu Jula

**Abstract**

The goal of the project is to establish whether a program position $P_{target}$ is reachable from a program location $P_{start}$ and, in the same time, infer the constraints that have to hold at $P_{start}$ for $P_{target}$ to be reachable and learn possible execution paths that lead to $P_{target}$ through $P_{start}$. The analysis is interprocedural and it uses Z3 SMT solver to decide the reachability and retrieve the constraints. The analysis is performed on the CFG of an application bytecode. The project can be implemented for LLVM or JVM bytecodes.

Our method consists mostly of symbolic execution. However, some static analysis is necessary for guiding the symbolic execution or filtering the outputs of the symbolic execution.

## Overview

Given the CFG in SSA form of an application bytecode, we generate a symbolic representation of the application. The symbolic program captures all the execution suffixes starting from up to $N$ call frames behind $P_{start}$, crossing $P_{start}$ and ending in $P_{target}$. We capture execution suffixes starting before $P_{start}$ in order to derive context information for $P_{start}$, and thus increase the precision of the reachability analysis.

For the reachability analysis, each statement $s$ from the CFG is labeled with a fresh boolean variable $l_s$ in the symbolic program denoting that $s$ has to be executed. By $\rho(s)$ we denote the symbolic representation of a statement $s$. For instance, the symbolic representation of the code

```
l: if (cond)
  l1: s1
else
  l2: s2
```

is

$$
\begin{aligned}
& l \rightarrow (\rho(cond) \wedge l_1 \vee \neg\rho(cond) \wedge l_2) \\
& \wedge \\
& l_1 \rightarrow \rho(s_1) \\
& \wedge \\
& l_2 \rightarrow \rho(s_2)
\end{aligned}
$$

## Learning reachability constraints

Let $Sym[Prog, N, P_{start}, P_{target}]$ be the symbolic representation of $Prog$, delimited by $N$, $P_{start}$ and $P_{target}$, and $l_{start}/l_{target}$ the labels corresponding to $P_{start}/P_{target}$.

Analyzing whether $P_{target}$ is reachable from $P_{start}$ is simply checking the satisfiability of $Sym[Prog, N, P_{start}, P_{target}] \wedge l_{start} \wedge l_{target}$. The constraints $C_{start \rightarrow target}$ under which $P_{target}$ is reachable from $P_{start}$ are given by the unsatisfiable core of $Sym[Prog, N, P_{start}, P_{target}] \wedge l_{start} \wedge \neg l_{target}$. We check the satisfiability using the Z3 SMT solver.

The constraints we need are the projection of $C_{start \rightarrow target}$ over the variables that are visible at $P_{start}$ and influence the outcomes of branches between $P_{start}$ and $P_{target}$.

## Learning possible execution paths

To actually retrieve the feasible execution paths ending in $P_{target}$ and crossing $P_{start}$, we need to combine symbolic execution with static analysis. Using static analysis, we find all the execution paths $l_1 : P_1, ..., l_n : P_n, l_{start} : P_{start}, l'_1 : P'_1, ..., l'_m : P'_m, l_{target} : P_{target}$. To see if such a path is feasible, we just check the satisfiability of $Sym[Prog, N, P_{start}, P_{target}] \wedge (\bigwedge_{1 \leq i \leq n} l_i) \wedge l_{start} \wedge (\bigwedge_{1 \leq i \leq m} l'_i) \wedge l_{target}$.

To find all the possible call stack suffixes ending in the method $M_{start}$ containing $P_{start}$, we just have to extract the call stacks from the feasible execution paths (trimmed at $P_{start}$) found above.

## Dealing with function calls and loops

To keep the approach scalable, we skip the symbolic execution of the function calls that do not lead to $P_{target}$. If the execution depends on their return values, then these calls are treated as uninterpreted functions in the symbolic program.

We can deal with loops in an iterative fashion. First, we unroll a loop once. If it is not sufficient (i.e., the *assume* statement after the unrolled loop makes the symbolic program unsatisfiable), we try to infer the correct number of iterations from the failed *assume*. For instance, if the symbolic execution indicated that $assume(i == 100)$ failed after an unrolled loop, we can learn that the loop has to be unrolled 100 times. If we can not infer the actual size of the loop, we double the size of the unrolled loop. After this learning step, we refine the symbolic program and repeat the symbolic execution. To do this efficiently, we could add the constraints encoding the new loop iterations on the fly, in order to make use of the incremental constraint solving.

If $P_{target}$ is inside a recursive function, we treat that function as a loop.

## Related work

Previous work exists in achieving immunity against exploits like buffer overrun. Systems like Vigilante [4], Bouncer [3], [2] or [1] use static analysis and/or symbolic execution to learn filters from existing exploits. Bouncer [3] also tries to learn new exploits that exercise the same vulnerability as the original exploit. The filters these approaches deduce refer only to the inputs of functions that process messages, and

they are applied only at the beginning of such functions. Moreover, these approaches only address exploits like buffer overruns.

We want to develop a technique similar to the above ones, but applicable in a larger context, to help addressing a larger range of problems. We want our approach to be able to infer meaningful predicates at arbitrary program positions; the predicates are expressed in terms of program variables visible at those positions. We will apply our approach to Dimmunix, to learn new deadlock signatures from an existing deadlock signature, and to learn conditions that have to hold right after acquiring a lock, for a deadlock location to be reachable.

In terms of handling loops in symbolic execution, only [1] describes a technique of handling loops, based on computing fix points. They first identify the induction variables in each loop. Then, they infer loop invariants in terms of induction variables and loop conditions. However, it is not clear how they infer the number of iterations starting from loop invariants. We adopt a simple technique that we believe it will work well in practice. Our technique is based on learning the size of a loop from symbolic execution failures caused by insufficient number of unrolled iterations. If we can not learn the size of the loop this way, we use widening techniques, as [1] uses.

# Discussion

## JVM vs. LLVM

1. Soot CFG vs. LLVM provided CFG.
2. Inlining filters at runtime in JVM vs. LLVM.
3. Building new call stacks for JVM vs. LLVM applications.

## Precision and Soundness

1. For multi-threaded programs, how do we guarantee that the learned predicates are sound, i.e., that they are not involved in data races between $P_{start}$ and $P_{target}$ ?
2. Since the analysis is bounded, and therefore imprecise, could we get too many possible call stacks ending in $M_{start}$ ?
3. Should we simulate two threads in the symbolic execution, to get more realistic signatures for Dimmunix? Otherwise, to get new signatures, we would just blindly combine call stacks ending in $M_{start}$.

# References


[1] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *In Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 2–16, 2006.

[2] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *ASPLOS*, 2008.

[3] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *SOSP*, 2007.

[4] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *SOSP*, 2005.