PASS Project Report Team 17

Amro Abdrabo, Lukas Friedlos

May 2022

1 Introduction

This document gives a high-level overview of the additions we made to `analyze.dl`. More detailed information about the implementation can be found as comments in the file.

2 Analysis

The main idea behind our analysis is to *explore execution traces of contracts* up to an arbitrary *fixed-sized depth*. In each step of the execution, taint is spread or removed by the semantics of the program, airing on the side of caution with regards to soundness of the analysis.

Some preliminary information needs to be computed in order to spread taint in a sound way:

Dependency Checking We capture all dependencies of of an SSA in the relation `ssa_depens_complete` with the following declaration:

```
.decl ssa_depens_complete(
    stmt: SSA, // Id of the statement
    args: number, // Bitvector encoding dependence on an argument
    sv: number, // Bitvector encoding dependence on a state variable
    sndr: number, // Boolean encoding dependence on the msg.sender
)
```

Throughout the analysis we use one-hot bitvector encoding for arguments and state variables. Each argument and variable are designated a specific bit of the bitvector. In the case of `ssa_depens_complete` the bitvectors `args` and `sv`, a set bit denotes a direct or indirect dependency on that argument/state variable.

Although this limits our analysis to programs which have at most 32 arguments and at most 32 state variables, we think this is a reasonable trade-off for simplicity and speed. Our analysis could be ported to a newer version of

Soufflé, where bitvector support has been added, removing this limitation while maintaining the benefits.

The relation is built recursively with base cases handling direct dependency, e.g. when an SSA is a load of a state variable. Indirect dependency is captured via a transitive closure under assignments and binary operations.

Exploring Execution Traces In order to explore all possible program executions, we built a finite state machine. It is finite because we limit the size of the explored traces to a fixed size. The reached states in this machine are captured the relation `program_run`. See its declaration here:

```
.decl program_run(
    last_blk: Block, // Last block in the execution trace
    sv_state: number, // Bitvector of state-variable taint
    arg_state: number, // Bitvector of argument taint
    guarded: number, // [0 | 1 | 2 | 3] denoting whether
                     // the state is privileged and/or guarded
    arg_stack: number,
    args_sndr: number, // Arguments which are semantically equivalent to
    // msg.sender (and all other dependencies are clean)
    sv_sndr: number // Same for state variables
    implicit_taint: number, // Implicit taint of the if block
    imp_block: Block, // the block which contains tainted if statement
    hist_size: number, // Current depth of the trace
    stack: BlkList, // Stack to pop from on function returns
)
```

The relation is initialized with a trace containing the constructor block only. Whenever program flow stops (such as after the constructor) we branch to the 'special block' for both the case where a privileged or unprivileged user calls the next function. From the 'special block' we branch to all possible function calls. Taint is soundly spread through `sv_state` and `arg_state` bitvectors.

We determine tainted sinks using the `program_run` relation. The relation defines the error states of our state machine. If such a state is computed in our execution, the relevant SSA is added to the relation.