Static Taint Analysis for Ethereum Contracts

Harish Rajagopal 20-946-349
Sven Grübel 15-923-493

Spring 2021

In this project, we implement a static taint analyser for Ethereum smart contracts. Our analyser takes as input an Ethereum smart contract written in Solidity and finds if an untrusted address can be used as the argument of `selfdestruct` in this contract.

1 Guards and Good/Bad Blocks

Guards are defined as boolean values that explicitly depend on `msg.sender` and all other values it depends on are untainted. Statements that are not a guard are modelled by the `notGuard(id, blk, stack)` relation. The reason why *not-guards* are tracked instead of *guards* is that there would be a cyclic dependency with negation if *guards* would be used directly, such that the Datalog relations can't be properly stratified.

Good blocks are defined as blocks where all incoming edges are protected by a guard. All other blocks are called *bad* blocks. All *bad* blocks are tracked by the analyser with the `argTaint(blk, stack)` relation.

To determine if a statement depends on `msg.sender`, the analyser tracks all statements and global variables which don't depend on `msg.sender` (again to remove negation out of a cyclic dependency).

2 Local Variable Taint

There are three sources for taint of local variables: `msg.sender` (called *sender-based taint*), user input and `msg.value` (called *conditional taint*), and previously tainted global variables (called *permanent taint*). *Conditional taints* and *sender-based taints* will be cleared after a guard condition, but *permanent taint* persists even through guards. We track it using `taintedLocal(id, blk, src, stack)`, which depends on the current block, the taint source, and the current call stack.

The reason for separating *sender-based taints* from *conditional taints* is that we want to identify taint sources other than `msg.sender` for `notGuard`. In all other cases, these two taint sources are treated the same.

Consider the following contract which shows the difference between these two taints for local variables:

```
contract Contract{
  address owner;
  int global_x;
  function foo(int x) {
    global_x = x;                 // global_x becomes permanently tainted after foo exits
  }
  function bar(int y) {
    x = global_x;                 // At this point, y is conditionally tainted and x is permanently tainted
    require(msg.sender == owner); // A guard is encountered
    selfdestruct(y);              // At this point, y is not tainted any more, but x is still tainted
  }
}
```

Our analyser correctly outputs "Safe".

3 Global Variable Taint

Taints for global variables have two states: *permanently tainted* (globals that can be tainted by an attacker), and *temporarily tainted* (for taints within a contract execution). Since *temporary taints* can change with every line, we pass around these taints between adjacent statements and blocks. We track *permanent taints* using `permaTainted(field)`. *Temporary taints* are tracked using `taintedGlobal(field, id, src, stack)`, which depends on the current statement, taint source (similar to local variable taints), and the current call stack.

If a global variable is *permanently tainted*, then a *temporary taint* is applied onto it at the start of a contract execution, with the taint source set to *permanent*. If a global variable has any *temporary taint* when a contract execution ends, then a *permanent taint* is applied onto that global.

The following example illustrates this behaviour:

```
contract Contract {
  int y;
  int z;
  address owner;
  function foo(int x) public {
    z = x;                      // z becomes temporarily tainted by x
  }                             // z becomes permanently tainted after foo exits
  function bar() public {       // z gets a temporary taint when bar starts, since it's perma-tainted by foo
    y = z;                      // y becomes temporarily tainted from z's temporary taint
    y = 0;                      // y is cleaned of its taint
    selfdestruct(address(y));   // safe, because y's taint no longer exists
  }
}
```

Our analyser correctly outputs "Safe".

4 Functions

To track function calls, call stacks are used. A call stack is essentially a list of jump transfer IDs. When a function is first called, the call stack is empty (i.e. `nil`). Whenever a function calls another function, the ID of the corresponding jump transfer is prepended to the call stack.

To track and ground valid stacks, we have the helper relations `validStack(stack)`, `validIdStack(id, stack)` and `validBlkStack(blk, stack)`. The latter two check if `id` and `blk` are possible with `stack`.

The inputs to a function can vary, and thus the call stack can identify which "state" the function's arguments (and global variables) are in. All local variables within a function get their relations, e.g. taints, either from function arguments or from global variables. Thus, almost all custom relations we define in Datalog depend on the call stack.

For example, consider the following contract:

```
contract Contract {
  address owner;
  function check(address x) public returns(bool) {
    // x will be tainted in call stacks [nil] and [bar->check, [nil]], but untainted in [foo->check, [nil]]
    return (msg.sender == x);       // whether check returns a guard depends on x's taint
  }
  function foo() public {
    require(check(owner));          // check is given an untainted value, so this is a guard
    selfdestruct(msg.sender);       // safe
  }
  function bar(address z) public {
    require(check(z));              // check is given a tainted value, so this is not a guard
    selfdestruct(msg.sender);       // vulnerable
  }
}
```

Our analyser correctly outputs "Tainted".

In accordance with the precision requirements in the project description, only stacks with a maximum depth of 3 nested function calls are tracked. If a call were to further increase the stack size, a special *stack overflow* token (`<so>`) is prepended to the call stack. Thus, all the jump transfers IDs after the first three IDs in such stacks will be replaced by just `<so>` — thereby being merged into one. This way, the analyser is able to mark local variables and global variables as tainted in all executions with a certain stack prefix if there is at least one tainted assignment with the same stack prefix, since their call stacks would be identical (due to `<so>`).

5 Sinks

Finally, to find out if a contract is vulnerable, we use the relation `tainted_sinks(id)`. This is set only if we have `selfdestruct(id, addr)` at this ID and `addr` is tainted, i.e. we have `taintedLocal(addr, blk, _, _)`, where `blk` contains `id`, for any taint source and call stack.