Static taint analysis for Ethereum contracts

Program analysis for system security and reliability

Eric Marty, Yannick Merkli

May 29, 2020

In this project we implement a static taint analyzer for Ethereum smart contracts written in Solidity. We want to label contracts as **Safe** or **Tainted** based on whether an untrusted user can call `selfdestruct` and therefore remove the contract from the blockchain.
More concretely we label a contract as **Tainted** if `msg.sender` is tainted when `selfdestruct` is invoked. Taints arise from function and `msg` arguments. A taint can be removed by a sanitizer, a branch (`if`, `require`, `assert`) depending on a guard. A guard is a statement, explicitly depending on `msg.sender` and all other values it depends must not be tainted.

1 Analyzer Structure

The solidity contracts are parsed and translated into a given Intermediate Representation (IR) and then analyzed in Datalog.

1.1 Function Context & Block Graph

To catch contextual relationships between functions, we implement `contexts`, based on a call stack, that allows us to assign taint and sanitized tags to statements, based on their function context. We initially build a call stack for each public function in the contract (`contexts.contextForInit`) and then recursively extend those contexts for internal function calls (`contexts.contextForCall`), up to a limit of 3 recursive function calls, to prevent infinite recursion.

```
contract Contract {
  address owner;
  function check(address x) public returns(bool) {
    return (msg.sender == x);      // x can be safe or tainted -> dep. on context
  }
  function foo() public {
    require(check(owner));         // owner is safe (constant)
    selfdestruct(msg.sender);      // safe
  }
  function bar(address x) public {
    require(check(x));             // x is tainted (function argument)
    selfdestruct(msg.sender);      // tainted
  }
}
```

We also build a `block_graph`, that represents the program flow inside of functions of a contract. This block graph provides a `dominatedBy` rule, that allows us to easily identify which blocks are always traversed to get to other blocks.

1.2 Taint

We implement two kinds of taint tags. We distinguish between hard taints (`is_tainted`) and weak taints (`maybe_tainted`) and their counterparts for storage variables `tainted_storage` and `maybe_tainted_storage`. Taint tags are propagated in the code on a per statement basis and are passed into and out of internal function calls.
We assign a hard taint tag to all function arguments in top level functions, any occurrences of `msg.sender` and `msg.value`, loads from tainted storage variables and return values from functions of context depth > 3. We assign a weak taint tag to elements, if they load from a weak tainted storage field.

```
contract Contract {
  address payable owner;
  function foo(uint256 x) public returns(uint256) {
    return 1;                        // safe return value
  }
  function kill() public payable {
    uint256 a = msg.value;           // a is hard tainted
    a = foo(msg.value);              // a is no longer tainted
    require(msg.sender == address(a)); // safe guard since a is not tainted
    selfdestruct(owner);             // safe
  }
}
```

We assign storage fields a hard/weak taint tag if they store a value that has a hard/weak taint assigned to it. Additionally we assign storage fields a weak taint tag on the first line in every top level function, if there exists a function where a storage field has a weak/hard taint tag assigned on the last line.

```
contract Contract {
  address payable user; address payable owner;
  function taintUser() public {
    user = msg.sender;           // user field gets hard tainted
  }
  function kill(int x) public {  // user field is weak tainted at start of kill
    owner = user;                // owner field gets weak tainted by user field
    address a = owner;           // a gets weak tainted by owner
    require(msg.sender == a);    // a is weak tainted
    selfdestruct(owner);         // tainted
  }
}
```

1.3 Sanitizer

We treat every guard condition that depends on `msg.sender` as a potential guard (`maybe_guard`), however we do not immediately check for taint tags on the other elements. Instead, we build a stack of `(SSA, Context)` tuples, which includes each statement the guard depends on (except `msg.sender`) and the context in which it was assigned. This is done in order to avoid cyclic redundancies between `sanitized` and `tainted` tags. A block which branches on a `maybe_guard` (`maybeGuardBlock`) forward propagates `maybeSantitizedBlock` tags into descendant blocks and descendant function calls. It further backpropagates `maybeSantitizedBlock` tags to all blocks of equal or lower branching depth.
We further handle functions with multiple returns as an edge case, since multiple returns allow for guard evasion. We assign `notSanitizedBlock` tags to blocks, to which multiple returns point to, of which at least one has no forward propagated `maybeSantitizedBlock` tag.

```
contract Contract {
  address payable owner;
  function maybeSanitize(int x) public {
    if(x < 5) {
      return;                     // evade guard by returning early
    }
    require(msg.sender == owner);  // guard
  }
  function foo(int x) public {
    maybeSanitize(x);             // guard, but can be evaded
    selfdestruct(msg.sender);     // tainted
  }
}
```

1.4 Tying Everything Together

A sink (`selfdestruct`) is tainted if the block it lies in is not a forward propagated `maybeSantitizedBlock`, the block is a forward propagated `maybeSantitizedBlock` but at least one program execution path does not pass a guard (`notSanitizedBlock`) or the block is a forward propagated `maybeSantitizedBlock` but the SSA stack of its guard, contains at least one tuple `(id, ctx)` where `id` is tainted in context `ctx`.