Homework 3 (due May 23)

In the third homework, you will implement an improved (context-sensitive) version of the Flow analysis from the second homework, as well as a small taint vulnerability analysis. Finally, you'll start running your analyses on the full set of contracts we have supplied. In this way, you can experimentally measure both precision and scalability. Optimization of Datalog code will start being essential.

1. Context-sensitive Flows: In the previous homework you defined the predicate `Flows(fromVar, toVar)`: there is information flow from variable `fromVar` to `toVar`. Most of you followed a simple and intuitive context-insensitive approach: at every call, you considered that there is an unconditional flow from the actual parameters of the call to the function's formal parameters, as well as from the forml return variable to the actual return variable at the call-site. This has the precision disadvantage that two different function calls get their flows merged. E.g., a function

   ```
   function id(x) { return x; }
   ```

   called at two program sites

   ```
   y1 = id(x1);
   ...
   y2 = id(x2);
   ```

   (under a context-insensitive analysis) will produce (among others) `Flows("x1", "y2")` and `Flows("x2", "y1")`, which do not hold.

   In this homework you will make `Flows` be context sensitive. The context will be the call site of every function, i.e., you will implement a 1-call-site-sensitive analysis. There are several ways to do this, but one could be:

   - Separate relations `LocalFlows(from,to)` and `ArgReturnFlows(context, from, to)`. The context is simply a basic block id (type `Block`), for the calling block.
   - The definition of `LocalFlows` appeals to that of `ArgReturnFlows` and vice versa.
   - There are two final relations `CSForwardFlows` and `CSBackwardFlows` that combine the above. The first is for all variable flows between variables in the same function, or in functions (transitively) called by that function. This is the precise context-sensitive relation that you will typically need. The second is a supplement to also catch the case of which callee variables may flow to which caller variables. This is more rarely used.

   (Note: As we will discuss in class, you could get the same effect with function summaries, since your `Flows` currently does not model storage. But the homework asks you to use context sensitivity explicitly, since it anticipates storage modeling in the next homework.)

   Examine the impact of this change experimentally.

2. Define the concept of tainted values and see if such a value can reach a sensitive operation. For the purposes of the homework, a tainted value is one that is retrieved at a `CALLDATALOAD` instruction and sensitive operations are `DELEGATECALL` and `CALLCODE`.

This definition admits many variations. E.g.,

- for precision (i.e., fewer but higher-confidence warnings): you can consider as tainted only values produced at a `CALLDATALOAD` not under some guard over who called the contract (instruction `CALLER`).
- for completeness (i.e., more warnings): you can define simple concepts for tainted values entering persistent or transient memory (Storage/Memory--instructions `SSTORE`/`SLOAD`, `MSTORE`/`MLOAD`). For instance, the whole memory can be considered tainted if a tainted value is written at any location. This can be considered either by extending `Flows`/`CSFlows` or by adding a new predicate, e.g., `FlowsIncludingMemory`.

Explore at least one of the above options (with the first being easy).

3. Apply your analyses to simple contracts you write but also to the contracts from gigahorse-benchmarks, through the `gigahorse.py` batch analysis runner. (Use `--help` for options.)

   What stats do you get? How many analyses time out? Optimize your code based on the guidelines for Datalog optimization we'll discuss in class.