Static Taint Analysis for Ethereum Contracts

Hugo Queinnec, Philippe Schlaepfer
Program Analysis for System Security and Reliability

June 10, 2022

1 Introduction

The goal of this project is to implement a static taint analyzer for Ethereum smart contracts. We have designed our analyzer using the architecture shown in Figure 1 below. In the following, we will explain each block of the flowchart and how they are linked. Each block represents a Datalog relation, but keep in mind that not all relations are shown on this flowchart. It has been simplified to show only the blocks that are conceptually important. A black arrow pointing from block A to block B means that block A is defined using block B. If the arrow is red, it is a negated relation.

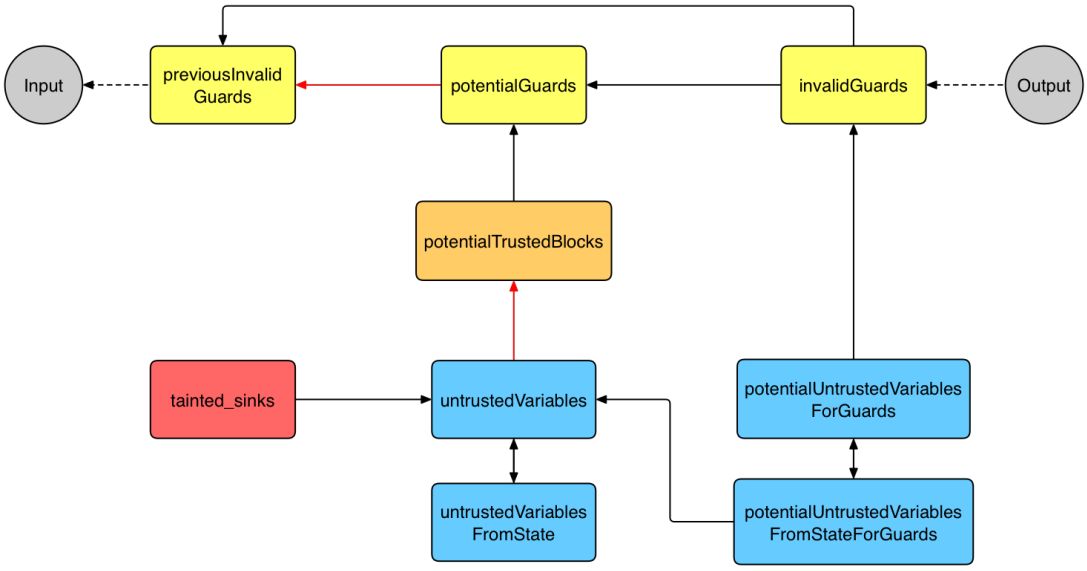

Figure 1: Simplified Architecture of our Analyzer

2 Detailed Architecture

2.1 A first iteration

2.1.1 Find potential guards

Initially, the block `Input` is empty, therefore the block `previousInvalidGuards` is also empty. The first step is to compute `potentialGuards`. `potentialGuards` is composed of the statements that depend on `msg.sender`, that are the condition of a branch, and that are not invalid (a guard is invalid if it is an untrusted statement). We will see later why `potentialGuards` uses `previousInvalidGuards` and not `invalidGuards`. As `previousInvalidGuards` is initially empty, we initially suppose that all potential guards are actual guards. In the following, the idea will be to propagate taints and to find all potential guards that are actually invalid.

2.1.2 Find potentially trusted blocks and untrusted variables

Once we have our `potentialGuards`, we can obtain `potentialTrustedBlocks`. These are the blocks that are protected by a potential guard. This is what allows us to compute `untrustedVariables`. Initially, untrusted variables are the function arguments of functions called by the user, as well as `msg.sender`. But these variables are untrusted only if

they are not protected by a guard, meaning used in a block that is part of `potentialTrustedBlocks`.
`untrustedVariables` also contains statements that load untrusted state variables (in `untrustedVariablesFromStates`). If a load is preceded by a store during the execution of the function call, then the load is untrusted if the store operation stored an untrusted variable. If the load is not preceded by a store, then it is untrusted if there exists one function that makes the state variable untrusted at the end of its execution. Otherwise, it is trusted.
Now that we have all base cases of `untrustedVariables`, we can add all statements that derive from them and are therefore also untrusted. This can come from an assignment, an operation, a transfer argument, etc.

2.1.3 Find tainted sinks

`tainted_sinks` are defined as `selfdestruct` statements that use an untrusted address. We output "Tainted" if there exists at least one tainted sink. But we considered (for the moment) that all potential guards were actual guards. Hence `tainted_sinks` is not correct yet: we still have to find guards that are invalid.

2.1.4 Find invalid guards

To find invalid guards, we need to find statements in `potentialGuards` that are untrusted.
We can't use `untrustedVariables` directly because it does not contain untrusted variables around the guards that we want to evaluate and that are potentially invalid. We solve this with `potentialUntrustedVariablesForGuards`, which is a larger set than `untrustedVariables`, and which does not take guards into account. As we use this set to evaluate guards, we do not include `msg.sender` as untrusted variable. This is because `msg.sender` is supposed to be trusted when evaluating a guard. For example, the guard `msg.sender == msg.sender` is valid. We use the block `potentialUntrustedVariablesFromStateForGuards` that plays the same role as `potentialUntrustedVariables-FromState` before.

2.2 The iterative process

So we now have our set of `invalidGuards`. This brings us back to the initial question: Why do `potentialGuards` use `previousInvalidGuards` and not `invalidGuards`? We can easily see on the flowchart that if we drew a negative relation from `potentialGuards` to `invalidGuards`, we would have a cyclic negation.
To avoid this, we use the following mechanism. Once we have found `invalidGuards`, we export it as a CSV file (this is the `Output` block on the flowchart). If we run the analysis a second time, the block `Input` will take this CSV file, which will fill the `previousInvalidGuards` block.
This iterative process allows to avoid the cyclic negation. We have to iterate several times in order to be sure to find all `invalidGuards`, because at each iteration, if `invalidGuards` changes, then `untrustedVariables` changes, which can in turn modify `invalidGuards` again.
In all of our tests, 4 iterations were always sufficient. However, to be entirely sure, we have decided to always perform 8 iterations. This iterative process is still really fast: it never takes more than 5 seconds to analyze a contract, and it takes 2.1 seconds on average on our 86 tests (running Soufflé natively on an Intel Core i5).

2.3 Context

A last important point to explain is the context. Almost all previous blocks, in addition to take a statement as argument, take a second argument called context. The context allows to differentiate if a function is called directly by the user (the context will be "`UCall`"), or if it is called by another function. In this last case, the context will be the statement provided as an argument to the function. This makes sense because the only thing that differentiates two function calls is the value of the function argument, and more specifically in our case: if it is trusted or not.

3 Conclusion

To conclude, our architecture is fast and works on all preliminary test cases.
However, working with custom tests, which are more complicated, we have found some limitations of our analyzer. We encountered several edge cases that did not work properly. In most cases, we were able to fix our analyzer to make it work, but we found 2 complex scenarios where our analyzer does not give a proper result and writing a fix would involve a major architectural change. In such edge cases, we went for a fix that categorize them as tainted, because soundness is more important than precision.