An Empirical Study of Implicit Information Flow

Cristian-Alexandru Staicu, TU Darmstadt

Advisor: Michael Pradel

SOLA SoftwareLab

TECHNISCHE UNIVERSITÄT DARMSTADT

Overview

Information flow analysis is widely used to enforce security policies that prevent sensitive data from flowing into untrusted sinks. For languages that are difficult to analyze statically, such as JavaScript, dynamic information flow analysis is popular. Unfortunately, dynamically analyzing implicit information flows, in particular those caused by not executing a particular branch, is non-trivial. Recent research has started to address this problem, but it requires additional work to become practical.

We present an empirical study, involving 28,614 lines of JavaScript code and over 20,000 executions, that addresses the following questions:

RQ1: How common implicit flows are in real-world JavaScript programs? Is it worthwhile to analyze them?

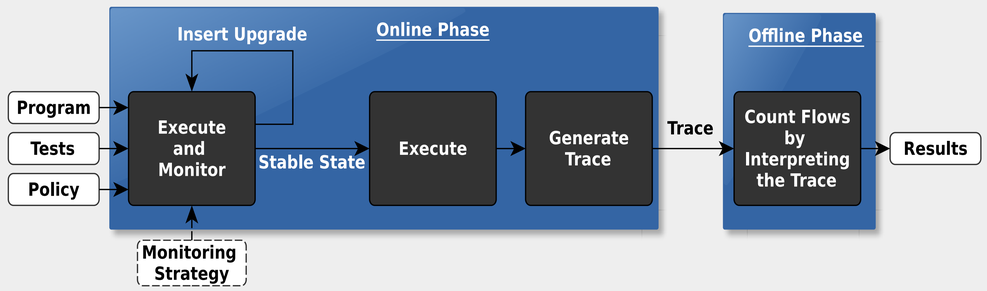
RQ2: How much do the different kinds of flows contribute to violations of security policies?

RQ3: What is the influence of the policy on the prevalence of different kinds of flows?

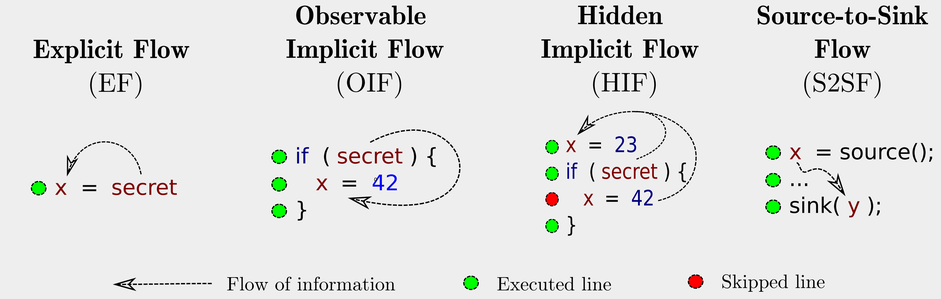
Possible Impact: The study shows that implicit flows, both caused by executing and by not executing a particular branch, are common. In particular, we find that an analysis that monitors only a subset of all kinds of information flows misses various violations of the checked security policy.

Methodology

- We use automatically inserted upgrade statements (in the manner of [1]) and randomly generated policies to observe different types of flows.



- *IFlow, a* custom language, is used to log all the information flow relevant events.



[1] Birgisson, Arnar, Daniel Hedin, and Andrei Sabelfeld. "Boosting the permissiveness of dynamic information-flow tracking by testing." Computer Security–ESORICS 2012. Springer Berlin Heidelberg, 2012. 55-72.

Example

- We present a simple code fragment and two runtime traces on which we highlight the different types of flows:

Source code with upgrades inserted

```
var gotIt = false;
markAsSource(getPasswd);
var passwd = getPasswd();
var paddedPasswd = "xx" + passwd;
var knownPasswd = null;
if (paddedPasswd === "xxxtopSecret") {
    gotIt = true;
    knownPasswd = passwd;
}
markAsSink(ajaxRequest);
ajaxRequest("evil.com", upgrade(gotIt));
```

Trace and graph representation for passwd = "topSecret"

```
-
-
source(0);
operation(1,0);write(1);
-
operation(2,1);push(2);
write(3,-1);
write(4,-1);
pop();
-
upgrade(5,3);sink(5);
```

1 Source-to-Sink,
2 EF, 2 OIF, 0 HIF

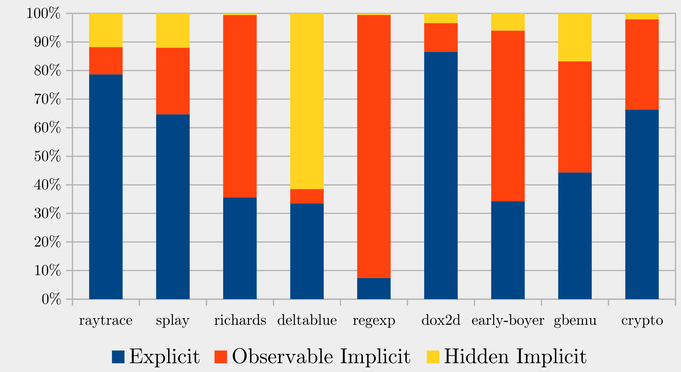
Trace and graph representation for passwd != "topSecret"

```
-
-
source(0);
operation(1,0);write(1);
-
operation(2,1);push(2);
-
-
pop();
-
upgrade(3,-1);sink(3);
```

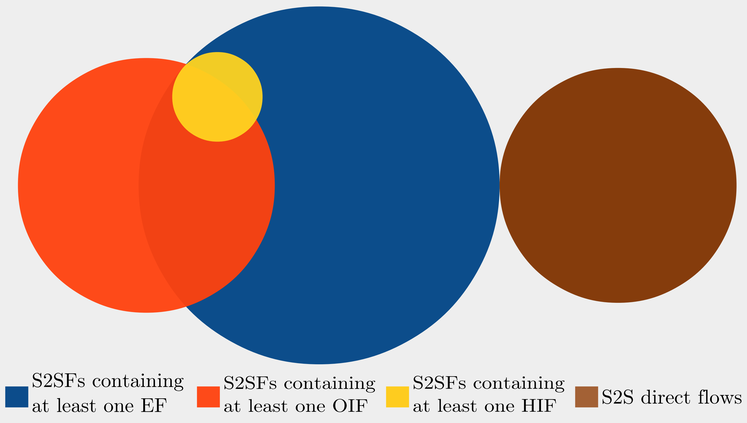
1 Source-to-Sink,
1 EF, 0 OIF, 1 HIF

Findings

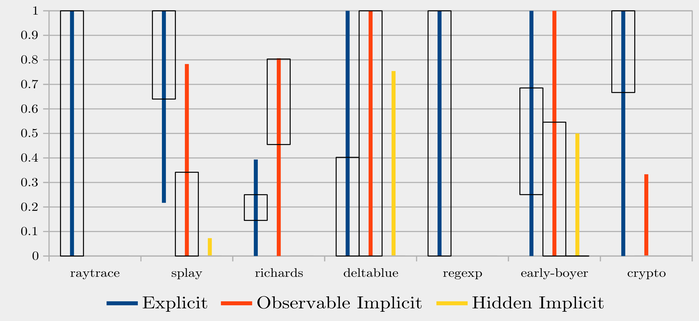
- We apply our methodology on Octane benchmarks:



RQ1 - All the three types of flows exist in every benchmark.



RQ2 – A source-to-sink flow has a 30% chance to depend on at least one implicit flow.



RQ3 - The prevalence of flows heavily depends on the policy.