SOLA SoftwareLab

TECHNISCHE UNIVERSITÄT DARMSTADT

Universität Stuttgart

Extracting Taint Specifications for JavaScript Libraries

Semmle

GitHub

Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, Michael Pradel

Abstract

We propose a technique for automatically extracting taint specifications for JavaScript libraries, based on a dynamic analysis that leverages the existing test suites of the libraries and their available clients in the npm repository. Due to the dynamic nature of JavaScript, mapping observations from dynamic analysis to taint specifications that fit into a static analysis is non-trivial. Our main insight is that this challenge can be addressed by a combination of an **access path mechanism** to name entry and exit points and the **use of membranes** around the libraries of interest.

We implement our ideas in a tool called TASER which we put to the test in a large-scale evaluation that answers the following research questions:

RQ1: Can TASER successfully extract specifications?
RQ2: How efficient is TASER?
RQ3: Are the extracted specifications useful?
RQ4: How does TASER compare to existing solutions?

Overall, we show that the TASER is effective and efficient at extracting taint summaries and these summaries can improve a commercial taint analysis.

Examples

Client Code

```
let userInput = {
        tempDir: "./path/to/dir",                    // source
        cacheDir: "./path/to/cache"
}
const _=  require("lodash");
const rimraf = require("rimraf");

let obj = _.forIn(userInput, function(value) {

 rimraf(value, function(err) {
   if (err)
     console.log(err);
   require("chilld_process").exec("rm * -rf " + value); // sink
 });

})
```

entry point
exit point

Library Code

```
const fs = require("fs");
function rmdir (p, options, originalEr, cb) {
  fs.rmdir(p, function (er) {                         // sink
    cb(er);
  });
}
module.exports = function rimraf(p, options, cb) {
  fs.lstat(p, function (er, st) {
    return rmdir(p, options, er, cb)
  });
}
```

tainted value

Taint Summaries

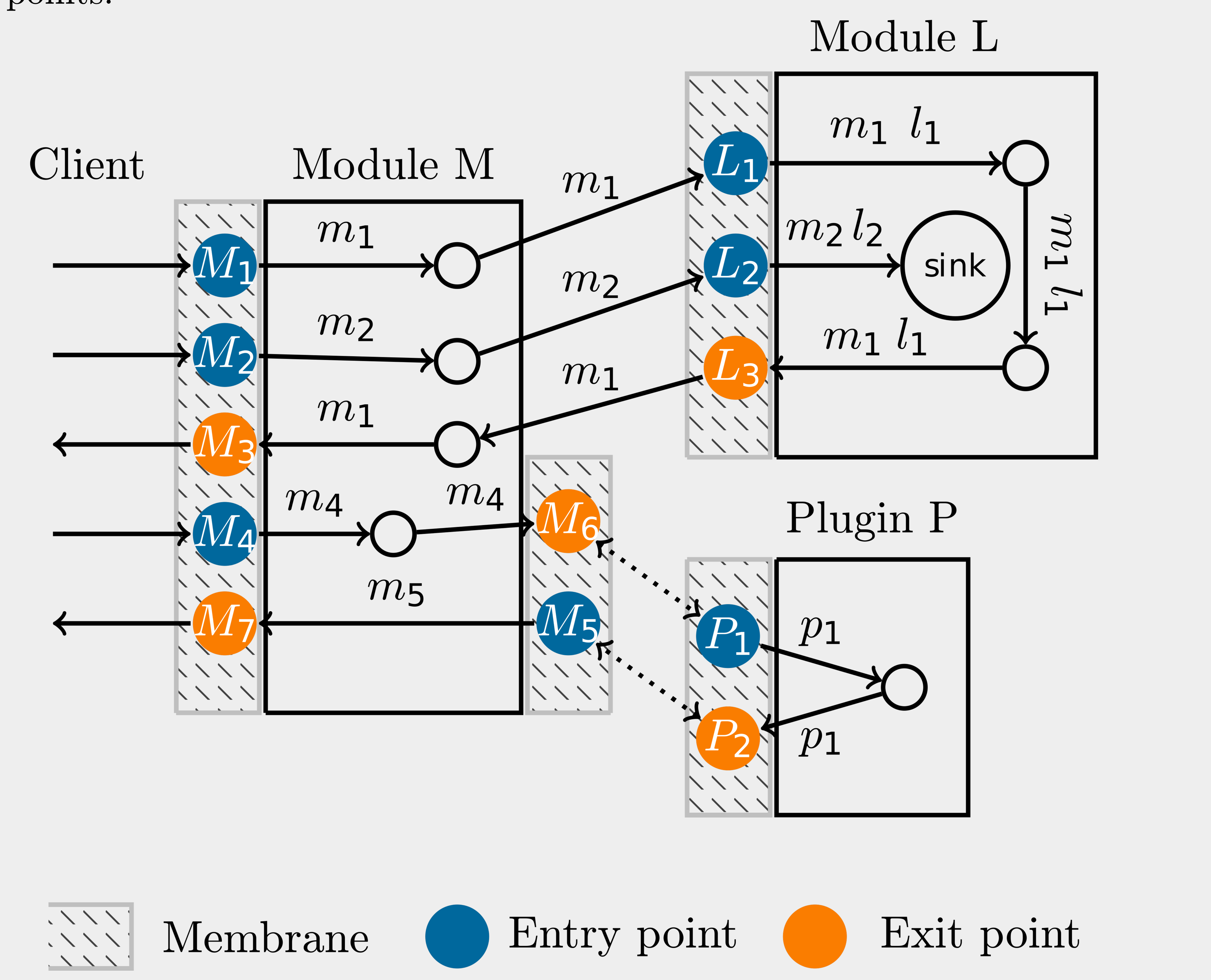
Additional sink:
(parameter 0 (root rimraf))
Propagation:
(parameter 0 (member forIn (root lodash)))
↓
(parameter 0 (parameter 1 (member forIn (root lodash))))

Membrane-based Analysis

A **membrane** is the set of all entry and exit points between two software components. Each reference that is exchanged through the membrane becomes part of it. Every point in the membrane is uniquely identified by an access path:

```
ap ::= (root  <uri>)
     | (member  <name>  <ap>)
     | (parameter  <i>  <ap>)
     | (return  <ap>)
     | (instance  <ap>)
```

TASER taints values at entry points in the membrane and declassifies values at exit points:



A taint summary is generated if a flow is detected:

- from an entry point to an existing sink (**additional sink**)
- from an existing source to an exit point (**additional source**)
- from an entry point to an exit point (**propagation**)

Results

Setup:

- **2300** npm modules
- **200** clients per module
- **10** minutes timeout
- **15,892** analyzed clients
- **5,707** clients with taint operations

RQ1: hundred of taint summaries

- **7,840** propagations
- **146** additional sinks
- **35%** non-trivial summaries

RQ2: 112 seconds per client

RQ3: LGTM produces new alarms when including specifications extracted by TASER.

Rule ID	New alerts
js/command-line-injection	2
js/file-access-to-http	64
js/path-injection	29
js/reflected-xss	5
js/regex-injection	13
js/remote-property-injection	20
js/user-controlled-bypass	2
js/xss	1

RQ4: Many security vulnerabilities are actually **undocumented additional sinks**. For example, advisory 27:

```
var printer = require("printer");
var benignInput = "printerName";
printer.printDirect({
   data: "Test",
   printer: benignInput,
   success: function (jobID) {
    console.log("sent to " + jobID);
   }
});
```

Additional sink inferred by TASER:
(member printer (parameter 0 (member printDirect (root printer))))