Dynamic Taint Analysis for Nodejs Applications

Siddharth Subramaniyam
Texas A & M University
College Station, TX
subsid@tamu.edu

Jeff Huang
Texas A & M University
College Station, TX
jeffhuang@tamu.edu

ABSTRACT

In this work, we develop a babel-plugin for adding domain specific instrumentation for Nodejs applications, that will help prevent malicious server side attacks such as SQL-injection, URL interpretation. The tool takes a rule-config file for a specific application and generates instrumented source code that can prevent tainted information flow through the application, at a domain level. The rule-engine supports adding rules by constraining code paths, validating inputs in a regular expression style language.

INTRODUCTION

Javascript has always been the goto language for frontend development. But since the release of Node.js in 2009, it has been rapidly gaining popularity in backend development, especially web servers. Its cross-platform ability, dynamic typing and rapid development combined with the fact that developers can write both backend and frontend in the same language has made it grow exponentially. But this also means its becoming more vulnerable and prone to attacks.

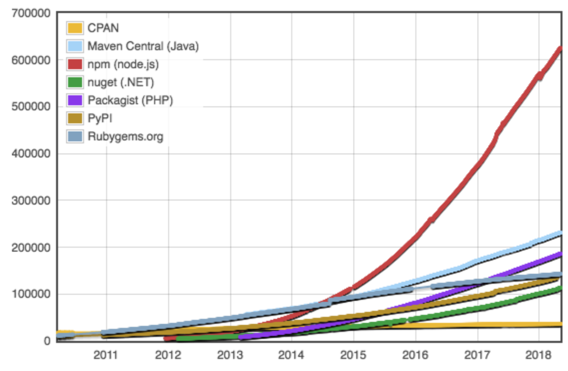
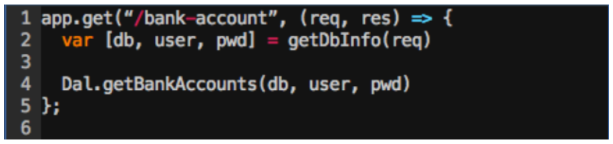


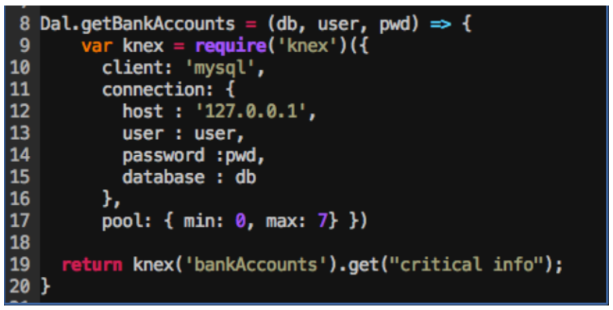
Figure 1. Rapid growth of Nodejs modules (https://modulecounts.com)

In this work, we develop a babel plugin that instruments javascript source based on domain specific rules. While we believe this can be used with any javascript backend, our initial version is geared towards web-servers. Since javascript is weakly typed, its hard to perform sound **static analysis**. On the other hand, performing **dynamic analysis** without any rules, will bloat the server code significantly, causing significant slowdown of the service. Our babel-plugin instruments javascript source code based on a rule-file, that a developer can create for each type of application. Our tool takes this rule file, and instruments certain code-paths in the source code *statically*, during the build phase.



```
app.get("/bank-account", (req, res) => {
  var [db, user, pwd] = getDbInfo(req)

  Dal.getBankAccounts(db, user, pwd)
};
```

Figure 2. A simple API endpoint that returns some critical information

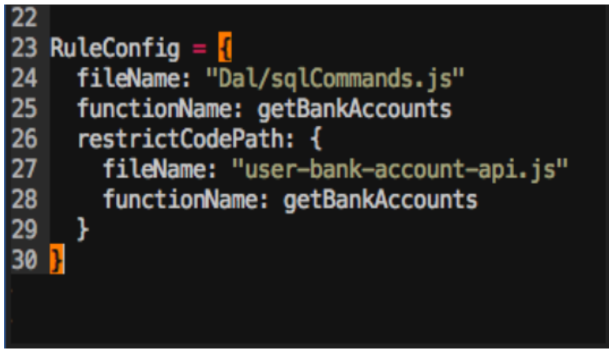


```
Dal.getBankAccounts = (db, user, pwd) => {
    var knex = require('knex')({
      client: 'mysql',
      connection: {
        host : '127.0.0.1',
        user : user,
        password :pwd,
        database : db
      },
      pool: { min: 0, max: 7} })

  return knex('bankAccounts').get("critical info");
}
```

Figure 3. A dataacess function that makes a database call

As an example, consider a simple webserver consisting of an *api endpoint* called */bank-account* (Fig.2), that makes a call to the database using a functions defined in the *Data Access Layer (DAL) called Dal.getBankAccounts* (Fig.3). Since this endpoint can return some critical information, it is useful to constrain the functions that can make calls to this endpoint. For example, if some other end-point is compromised, we can prevent that from making calls to this function. This can be done using a simple constraining rule as shown in Fig.4.

Our initial idea was to implement our tool using Jalangi2. Jalangi2 uses a techniques for shadowing various calls/variables in the code and perform checks on these shadow objects, this causes a 20-30x slowdown. Its hard to selectively instrument code using jalangi. Jalangi2 (the newer release by samsung) is also not actively maintained (Last commit at the time of writing this paper was April 2017, more than an year old). These reasons make it hard to integrate a plugin using Jalangi, into existing continuous deployment pipelines. Also, testing the instrumented code becomes hard. For our



```
RuleConfig = {
  fileName: "Dal/sqlCommands.js"
  functionName: getBankAccounts
  restrictCodePath: {
    fileName: "user-bank-account-api.js"
    functionName: getBankAccounts
  }
}
```

Figure 4. A simple constraining rule that prevents malicious calls to the above mentioned DAL function

tool, we decided to implement it as a babel plugin. The reason we chose to implement the tool as a babel-plugin was the prevalence of babel in the ecosystem. **Babel** is a javascript compiler, that allows developers to use modern javascript capabilities, without worrying about runtime (Nodejs) or browser compatibilty issues. Babel has support for the latest version of JavaScript through syntax transformers. These transformenrs are written as plugins and allow developers to use new syntax, without waiting for browser support. Due to its popularity, it has great documentation and active support. Babel uses a javascript parser called *babylon* and makes it easy to modify the syntax tree for instrumentation.

RELATED WORK

Dynamic Analysis

Taint analysis is essentially analysis of running code, by tracking information flow. In dynamic taint analysis [1], we define a taint source, sink and information flow rules, to track taint flow. In our domain specific tool, the rules, sources and sinks are defined in the rule-file (albeit, in a simpler language). There has been a lot of work in taint analysis [2, 3, 4] , but very few in javascript [5].

Javascript taint analysis

To our knowledge, there hasn't been much work in *domain specific javascript taint analysis* in the literature, but there are a few opensource projects that do the same. Jalangi is an execellent framework for dynamic analysis in general [5], but is not meant to be used for domain specific analysis, or production deploys.

METHODOLOGY

Our implementation makes use of the babel-runtime and runs in the transform stage of the babel compiler. Babel is a generic multi-purpose compiler for JavaScript. The babel compiler takes source code as input and generates instrumented/transformed code. The 3 main stages (Fig. 5) in a babel run are 1) Parse 2) Traverse and Transform 3) Code generation. Each of these steps involve creating or working with an Abstract Syntax Tree or AST.

For example, using babylon, a simple piece of javascript code such as Fig. 6 can be represented as Fig. 7, in a tree structure similar to ESTree. This makes it easy to transform source code, by modifying nodes in the tree.

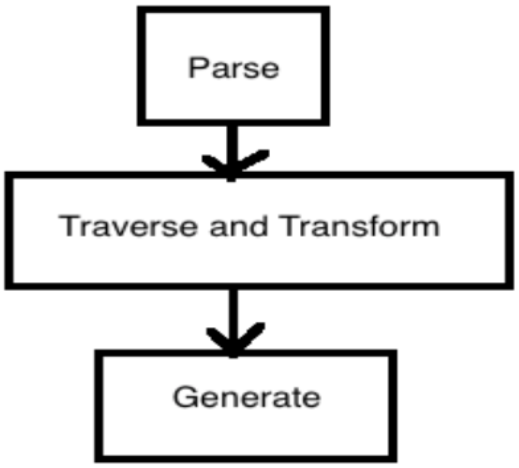
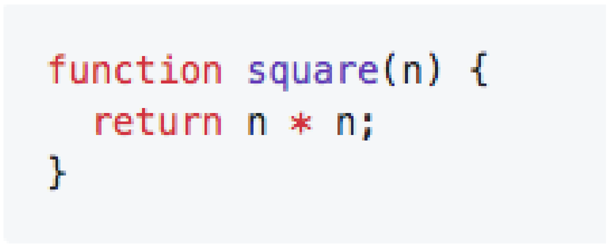


Figure 5. Stages in Babel



```
function square(n) {
  return n * n;
}
```

Figure 6. Simple javascript function

Parse

The parse stage, takes code and outputs an AST. There are two phases of parsing in Babel: **Lexical Analysis** and **Syntactic Analysis**.

Lexical Analysis

Lexical Analysis will take a string of code and turn it into a stream of tokens. This is done by the babylon parser, which babel uses.

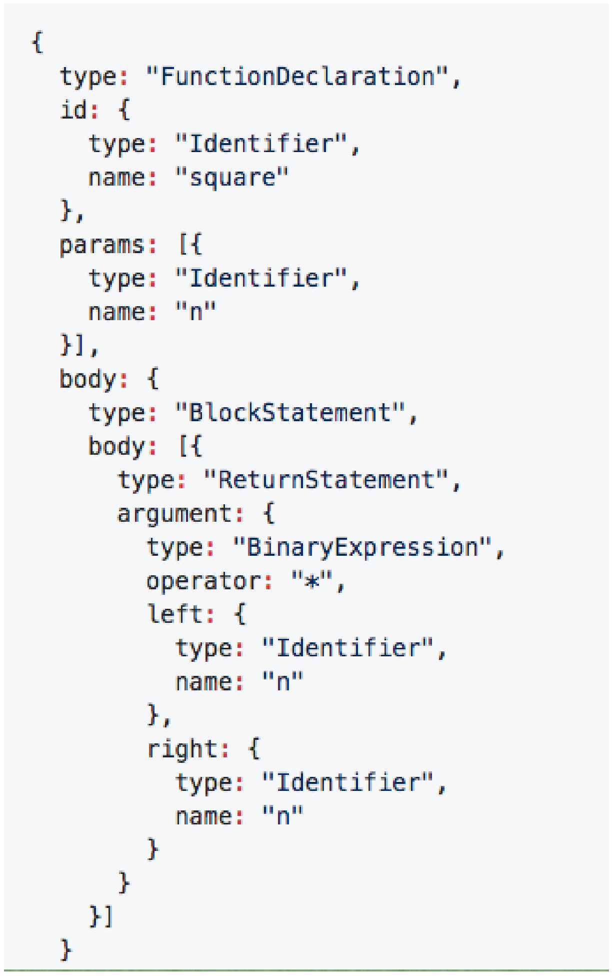
Syntactic Analysis

Syntactic Analysis will take a stream of tokens and turn it into an AST representation. Using the information in the tokens, this phase will reformat them as an AST which represents the structure of the code in a way that makes it easier to work with.

Transform

When you want to transform an AST you have to traverse the tree recursively. Babel handles this traversal and transformation using the **visitor pattern**. The idea is to write *visitor functions* for nodes we would like to perform transformation on. For our tool, we implement various visitor functions based on the rule-config the developer provides. This traversal process happens throughout the Babel transform stage and is recursive.

An AST generally has many Nodes, which makes handling relationship between nodes important. Babel provides **paths**,



```
{
  type: "FunctionDeclaration",
  id: {
    type: "Identifier",
    name: "square"
  },
  params: [{
    type: "Identifier",
    name: "n"
  }],
  body: {
    type: "BlockStatement",
    body: [{
      type: "ReturnStatement",
      argument: {
        type: "BinaryExpression",
        operator: "*",
        left: {
          type: "Identifier",
          name: "n"
        },
        right: {
          type: "Identifier",
          name: "n"
        }
      }
    }]
  }
```

Figure 7. Generated ESTree Object

an abstraction for a giant mutable object, that can be modified by various visitors.

Generate

The code generation stage takes the final AST and turns it back into a string of code, also creating source maps.

Code generation is pretty simple: you traverse through the AST depth-first, building a string that represents the transformed code. Babel handles this.

Thus, most of code instrumentation is done in the transform stage.

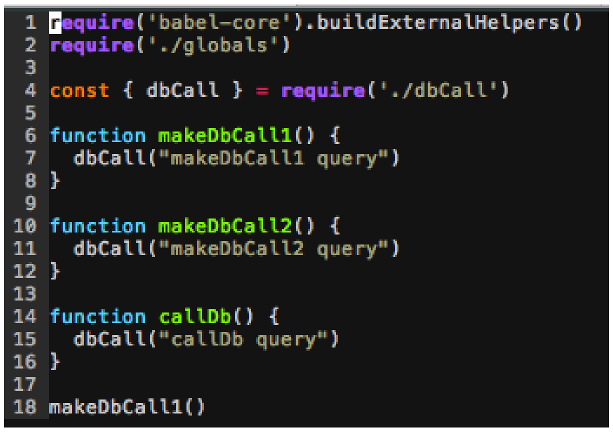
THE RULES FILE

The rule config file, is a simple JSON file, that provides configuration information for a particular application. Currently, we support two types of rules 1) RestrictAccess rule 2) Input validation rule. But the plugin supports adding new rule types with ease.

For the initial version, we chose the language to be a simple regular-expression style language, that will allow the rules to use regexp for matching function names/file names.

RestrictAccess type rules

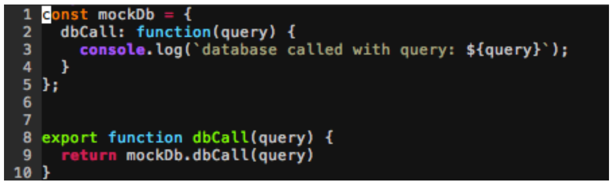
These type of rules help prevent tainted information flow to certain functions, by allowing the developer to constrain information flow paths. For example, consider 3 function as shown in Fig. 8 that call a common function *dbCall* (Fig. 9). A restrict access type rule can constrain the *dbCall* function to only execute when its *caller* is defined by the developer. Thus, even if some API is compromised, these types of restrictions can prevent access to certain critical parts of the system.



```
require('babel-core').buildExternalHelpers()
require('./globals')

const { dbCall } = require('./dbCall')

function makeDbCall1() {
  dbCall("makeDbCall1 query")
}

function makeDbCall2() {
  dbCall("makeDbCall2 query")
}

function callDb() {
  dbCall("callDb query")
}

makeDbCall1()
```

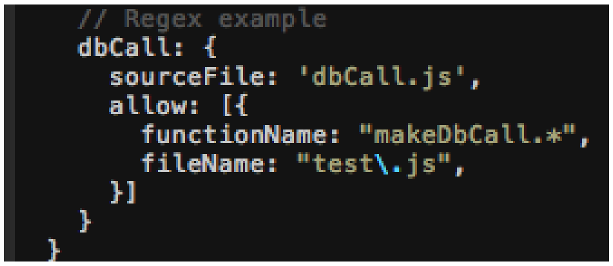
Figure 8. 3 functions calling a common function

Fig. 10 shows a simple restriction rule, that only allow functions that start with *makeDbCall* and present in *test.js* to call the *dbCall* function defined in *dbCall.js*. The filename restriction provides a additional layer of protection, so that the malicious user cannot just change their calling functions name.



```
const mockDb = {
  dbCall: function(query) {
    console.log(`database called with query: ${query}`);
  }
};


export function dbCall(query) {
  return mockDb.dbCall(query)
}
```

Figure 9. dbCall function that can return critical information

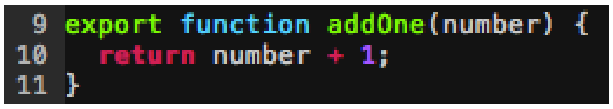


```
  // Regex example
  dbCall: {
    sourceFile: 'dbCall.js',
    allow: [{
      functionName: "makeDbCall.*",
      fileName: "test\.js",
    }]
  }
}
```

Figure 10. RestrictAccess type rule

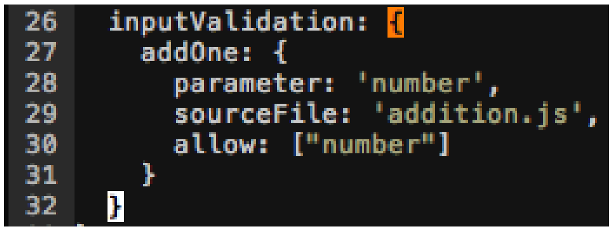
Input Validation type rules

Input validation type rules provide simple type validation to parameters of functions. Since javascript is weakly typed, often times, functions getting called with unexpected types leads to program crashes or undesired behavior. Consider the function *addOne* (Fig. 11, it expects the parameter *number* to be numeric, this can easily be coded using an input validation rule. Any other type of parameter would prevent this function from running. This can be easily validated using a rule such as Fig. 12.



```
export function addOne(number) {
  return number + 1;
}
```

Figure 11. Simple addOne function that expects a numeric parameter



```
  inputValidation: {
    addOne: {
      parameter: 'number',
      sourceFile: 'addition.js',
      allow: ["number"]
    }
  }
```

Figure 12. RestrictAccess type rule

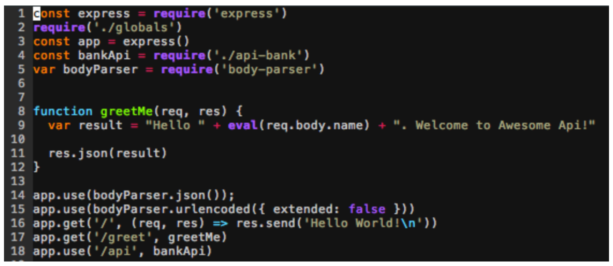
TESTING WITH NODEJS SERVER

For testing our application, we created an expressjs (Popular nodejs web server) application with a few endpoints and various code paths. We tested the behavior of the application with and without each rule we used for testing.

The example application we created has 2 endpoints 1) */bank-account* 2) */greet*.

The bank-account endpoint returns some critical information and hence should be restricted. (Fig. 15 16)

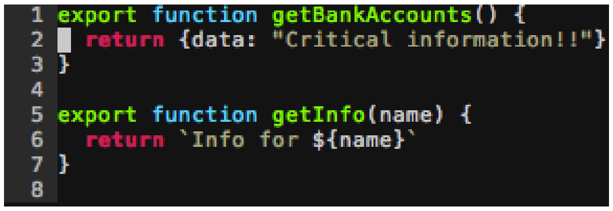
The greet function (Fig. 13) is a simple endpoint that greets a user with the given name. Since the greet function does an *eval*, it is possible for a malicious user to execute some javascript using the greet end-point. For example, one can make a call to the *getBankAccounts* function using the greet api as shown in Fig. 17.



```
const express = require('express')
require('./globals')
const app = express()
const bankApi = require('./api-bank')
var bodyParser = require('body-parser')


function greetMe(req, res) {
  var result = "Hello " + eval(req.body.name) + ". Welcome to Awesome Api!"

  res.json(result)
}

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }))
app.get('/', (req, res) => res.send('Hello World!\n'))
app.get('/greet', greetMe)
app.use('/api', bankApi)
```

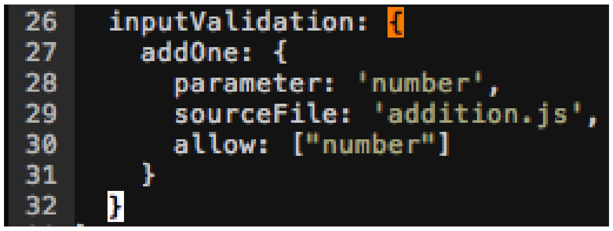
Figure 13. The greetMe function does an eval on the user parameter without any validation

This can be avoided by restricting critical functions to be called by only certain allowed functions defined in the domain. This is what is done by the developer in the rules file. (Fig. 19) In this example, we restirct access to the *getBankAccounts* function, so that only *getBankData* defined in *api-bank.js* can call it. Doing so, prevents */greet* from injecting javascript code and calling *getBankAccounts*. (Fig. 18).



```
export function getBankAccounts() {
  return {data: "Critical information!!"}
}

export function getInfo(name) {
  return `Info for ${name}`
}
```

Figure 14. getBankAccounts is a mock function that can be thought of as returning some critical information



```
  inputValidation: {
    addOne: {
      parameter: 'number',
      sourceFile: 'addition.js',
      allow: ["number"]
    }
  }
```

Figure 15. Bank accounts API

The babel plugin parses the rules file, and instruments only the */getBankAccounts* function. Thus, the final output of the babel adds minimal overhead to the existing code, and does not use the rules file after the static instrumentation.

The plugin instruments the source code such that, only if the allowed function is present in its stack, will it execute the restricted function. This helps protect certain endpoints.

Since the plugin only instruments endpoints that have rules, this makes the code maintain performance, compared to instrumenting all call sites.

FUTURE WORK

We would like to measure the performance impact by taking a real-world nodejs application and instrumenting it using our rules. It would be interesting to empirically assess the impact of this kind of selective instrumentation.

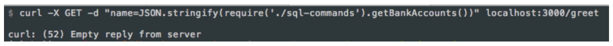
We only support regular expressions in this initial release. In the future, we would like to support a more powerful and flexible DSL that makes specifying rules a lot easier.

It would also be useful to provide intelligent defaults for a certain class of applications. For example, any CRUD based API can have a standardized structure for code organization and workflows. These can be enforced by the rules file.

Furthermore, we would like to support domain specific code styles, that can be enforced this way. For example, to make sure developers don't take 'shortcuts' and call restricted functions without using an api, we can include first-class rules that handle such calls. Though, this may not make sense for an production app, it can certainly be enforced in a staging environment.

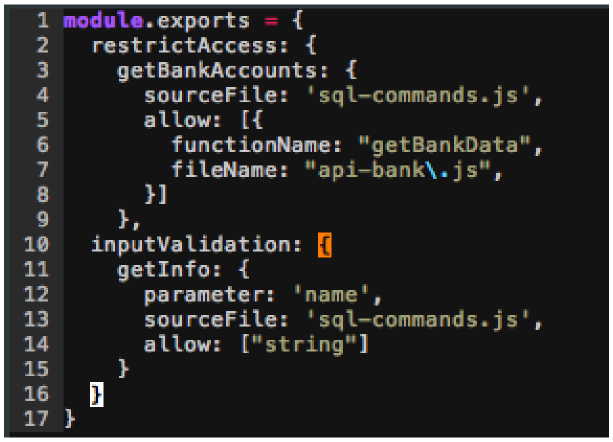
CONCLUSION

In this work, we have developed a tool for dynamic taint analysis of nodejs applications, by selective instrumentation using a babel plugin. The widespread use of javascript increases the need for more analysis tools in its ecosystem.



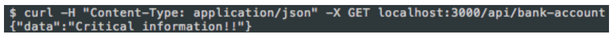
```
$ curl -X GET -d "name=JSON.stringify(require('./sql-commands').getBankAccounts())" localhost:3000/greet

curl: (52) Empty reply from server
```

Figure 18. Call to greet with RestrictAccess rule gives no response



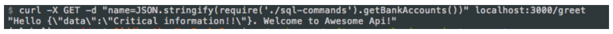
```
module.exports = {
  restrictAccess: {
    getBankAccounts: {
      sourceFile: 'sql-commands.js',
      allow: [{
        functionName: "getBankData",
        fileName: "api-bank\.js",
      }]
    },
  inputValidation: {
    getInfo: {
      parameter: 'name',
      sourceFile: 'sql-commands.js',
      allow: ["string"]
    }
  }
}
```

Figure 19. Simple rules that constrains access to getBankAccounts



```
$ curl -H "Content-Type: application/json" -X GET localhost:3000/api/bank-account
{"data":"Critical information!!"}
```

Figure 16. Call to /bank-account api



```
$ curl -X GET -d "name=JSON.stringify(require('./sql-commands').getBankAccounts())" localhost:3000/greet
"Hello {\"data\":\"Critical information!!\"}. Welcome to Awesome Api!"
```

Figure 17. Malicious call to greet api

Domain specific rules, help capture tainted information flow, by implicitly defining sources, sinks and flow-functions using rules.

Our tool has been implemented as a babel plugin, since babel is prevalent in the JS ecosystem and makes it easy to modify ASTs at the build stage.

We aim to extend the application to support more rule types and also provide a more powerful language for specifying rules.

REFERENCES

1. G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," in *ACM Sigplan Notices*, vol. 45, pp. 1–12, ACM, 2010.
2. J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.
3. E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security and privacy (SP), 2010 IEEE symposium on*, pp. 317–331, IEEE, 2010.
4. J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*, pp. 196–206, ACM, 2007.
5. K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, (New York, NY, USA), pp. 488–498, ACM, 2013.