

# JSTAP: A Static Pre-Filter for Malicious JavaScript Detection

Artifact: Samples, Scripts and Examples

By Aurore Fass

## ABSTRACT

In our paper<sup>1</sup>, we propose JSTAP, a modular static JavaScript detection system. Our detector is composed of ten modules, including five different ways of abstracting code (namely tokens, Abstract Syntax Tree (AST), Control Flow Graph (CFG), Program Dependency Graph considering data flow only (PDG-DFG), and PDG), and two ways of extracting features (leveraging n-grams, or Identifier values). Based on the frequency of these specific patterns, we train a random forest classifier for each module. Section 2 of our paper explains and illustrates how we built the previous ten modules, while sections 3.2, 3.3, and 3.4 leverage the different modules for an accurate malicious JavaScript detection.

Our artifact consists of the source code of JSTAP, along with 5,000 SHA1-unique samples from the datasets we used in our paper (section 3.1.1 and 3.1.2, please note that some samples do not belong to us and we are not allowed to give them away) and four scripts to automatically check JSTAP’s functionalities. In particular, the artifact contains the source code to build the different data structures we used (e.g., AST, CFG...), to train a random forest classifier on our JavaScript samples (including the features selection process with  $\chi^2$ , c.f. paper sections 2.2.3 and 3.1.3) and to classify (unknown) JavaScript inputs (also part of the artifact, along with the scripts to automatically launch the previous functionalities).

To test our artifact, 4 steps are needed:

- On Ubuntu 18.04, install nodejs, esprima, escodegen, python3, and python3-pip along with specific modules: we give more details, as well as the tested versions, in README.md;
- Run our script (provided) to generate the PDGs of the provided JavaScript files;
- Run our script (provided) to train a random forest model for a specific JSTAP module and on the selected JavaScript files (provided);
- Run our script (provided) to classify (unknown) JavaScript inputs (provided).

We give examples in the remainder of this document.

## 1 DOCUMENTATION

We host 5,000 samples (2,500 benign / 2,500 malicious) at <https://swag.cispa.saarland/files/JStap-additional.zip>. Specifically, for each class, we randomly selected 1,000 samples from our public datasets (GeeksOnSecurity (GoS), Hynek Petrak (Hynek) and Tranco top 10,000 websites, c.f. paper sections 3.1.1 and 3.1.2) for the classifier training (section 3.1.3), 500 samples for the features selection process (sections 2.2.3 and 3.1.3) and 1,000 for the actual classification (sections 3.2, 3.3, and 3.4). There are no duplicates among these 5,000 samples (based on their SHA1-sum). At this URL, we also host the different scripts to launch for testing JSTAP implementation.

### 1.1 Setup

Please clone our GitHub repository (<https://github.com/Aurore54F/JStap>) and refer to the *README.md* to install the different dependencies and requirements.

### 1.2 Samples and Scripts

Please download the samples and scripts from <https://swag.cispa.saarland/files/JStap-additional.zip>, unzip the file (password: JStap) and move the two folders (*scripts* and *samples*) at the root of our GitHub project you previously cloned. You should now have four subfolders in the *JStap* folder: *classification*, *pdg\_generation*, *scripts* and *samples*.

### 1.3 PDGs Generation

To test the PDG generation module, generate the PDGs of the *samples/Bad-validate* folder by running the shell script *generate\_pdg.sh* directly from the *scripts* folder:

```
cd scripts
./generate_pdg.sh
```

You will see a list of the currently stored PDGs appear. The PDGs will automatically be saved in the *JStap/samples/Bad-validate/Analysis/PDG* folder. Estimated duration: 5 min. We measured the estimated duration on a virtual machine with one core Intel(R) Core(TM) i5-7287U CPU at 3.30GHz. For time constraints, we already generated the PDGs of the five other samples folders.

### 1.4 Learning: Building a Model

To learn a model to detect future unknown malicious JavaScript documents, choose the level of the analysis among ‘tokens’, ‘ast’, ‘cfg’, ‘pdg-dfg’ and ‘pdg’ and the features among ‘ngrams’ and ‘value’. Then run the shell script *train.sh* directly from the *scripts* folder with the analysis level as first argument (e.g., *ast*) and the features used as second (e.g., *ngrams*):

```
cd scripts # if you are not already there
./train.sh ast ngrams
```

The model will automatically be saved in the *JStap/Analysis/Model* folder. Estimated duration: 8-10 min per module, except for the tokens with 40-50 min<sup>2</sup>.

### 1.5 Classification of Unknown JS Samples

To test the previous model on JavaScript documents, keep the same analysis level and the same features as for the model previously built. Then run the shell script *classify.sh* directly from the *scripts* folder with the analysis level as first argument (e.g., *ast*) and the features used as second (e.g., *ngrams*):

```
cd scripts # if you are not already there
./classify.sh ast ngrams
```

<sup>1</sup><https://swag.cispa.saarland/papers/fass2019jstap.pdf>

<sup>2</sup>Our VM was really struggling with the calls to the Esprima tokenizer

At the end, you will see a list of the files classified along with JSTAP predictions (benign/malicious). As mentioned in our paper sections 3.2.1 and 3.2.2, some files may not be analyzed; hence, they will not appear in the list. Estimated duration: 4-5 min per module, except for the tokens with 15-20 min.

## 1.6 Testing JSTAP Ten Modules

Finally, if you would like to test all ten JSTAP modules (i.e., the five analysis levels combined with the two possible features), run the shell script `all.sh` directly from the *scripts* folder:

```
cd scripts # if you are not already there
./all.sh
```

The ten models will automatically be saved in the *JStap/Analysis/Model* folder, while the predictions of JSTAP will

automatically be saved in the *JStap/Analysis/Results* folder. Estimated duration: 4 hours.

## 1.7 Disclaimer

Depending on the scripts being analyzed and the Esprima tokenizer/parser version, Esprima might throw an error if it cannot tokenize/parse the considered script(s) (for example if it does not recognize a specific construct). Still, at most a few scripts should be impacted by that.

For this artifact, we provide a small subset of our dataset; in particular, the training sets are significantly smaller than in our paper. Therefore, the detection results might slightly differ from the paper's.