GuardiaML: Machine Learning-Assisted Dynamic Information Flow Control

Scull Pupo, Angel Luis; Nicolay, Jens; Efthymiadis, Kyriakos; Nowe, Ann; De Roover, Coen; Gonzalez Boix, Elisa

Published in:
Proceedings of the 26th International Conference on Software Analysis, Evolution, and Reengineering (SANER 2019)

DOI:
10.1109/SANER.2019.8667979

Publication date:
2019

License:
Unspecified

Document Version:
Final published version

Link to publication

Citation for published version (APA):
Scull Pupo, A. L., Nicolay, J., Efthymiadis, K., Nowe, A., De Roover, C., & Gonzalez Boix, E. (2019). GuardiaML: Machine Learning-Assisted Dynamic Information Flow Control. In E. Shihab, D. Lo, & X. Wang (Eds.), *Proceedings of the 26th International Conference on Software Analysis, Evolution, and Reengineering (SANER 2019)* (pp. 624-628). [8667979] (SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering). IEEE. https://doi.org/10.1109/SANER.2019.8667979

GuardiaML: Machine Learning-Assisted Dynamic Information Flow Control

Angel Luis Scull Pupo*, Jens Nicolay*, Kyriakos Efthymiadis†,
Ann Nowé†, Coen De Roover*, and Elisa Gonzalez Boix*
*Software Languages Lab
Vrije Universiteit Brussel, Belgium
firstname.lastname@vub.be
†Artificial Intelligence Lab
Vrije Universiteit Brussel, Belgium
firstname.lastname@vub.be

***Abstract*—Developing JavaScript and web applications with confidentiality and integrity guarantees is challenging. Information flow control enables the enforcement of such guarantees. However, the integration of this technique into software tools used by developers in their workflow is missing.**

In this paper we present GuardiaML, a machine learning-assisted dynamic information flow control tool for JavaScript web applications. GuardiaML enables developers to detect unwanted information flow from sensitive sources to public sinks. It can handle the DOM and interaction with internal and external libraries and services. Because the specification of sources and sinks can be tedious, GuardiaML assists in this process by suggesting the tagging of sources and sinks via a machine learning component.

***Index Terms*—JavaScript Security, Programming Languages, Information Flow Control, Machine Learning**

I. Introduction

In applications that deal with information that is sensitive in nature, it is important to keep track of which information flows where to detect and prevent information leaks. This is especially so for web and cloud applications, which orchestrate different internal and external services by exchanging information to implement the application's functionality. Information flow control (IFC) can be used to enforce data confidentiality and integrity guarantees over application data [1]. However, IFC is challenging in any non-trivial application context. In particular, for web and cloud applications written in JavaScript, we identify the following specific challenges:

- How to deal with the various services with which the application interacts, and of which the source code may or may not be available?
- How should DOM nodes be handled?
- How can sources and sinks of information be identified, especially in large applications that interact with many different services?

In this paper, we present GuardiaML, a novel tool for dynamic IFC that uses machine learning for identifying potential sources and sinks of information flows. The goal of GuardiaML is to enforce IFC in JavaScript web and cloud applications. This is achieved in two steps. First, a developer can annotate sources and sinks in a JavaScript source file, assisted by recommendations from the tool. After annotations are added, GuardiaML can produce an instrumented version of the source code that can be executed instead of the original code. The instrumentation performs the actual tagging and tracking of data, and only changes the external behavior of the program w.r.t. the original when an information control policy is violated.

The foundation of GuardiaML is Gifc [2], a security policy specification language and library for dynamically enforcing policies. Gifc handles a large set of JavaScript features including the DOM, and enables the modeling of libraries and web APIs. GuardiaML extends Gifc with a machine learning component to statically detect potential sources and sinks. The algorithm used by this component was trained using the Node.js API as a dataset in which we manually annotated methods as sources or sinks. Supervised machine learning was performed to let the algorithm learn how to correctly classify sources and sinks to their respective class, and to show that the algorithm can generalize to unseen examples.

II. Background

A. Information Flow Control for Web Applications

Informally, information flow control enables preventing unwanted flows from sensitive *sources* of information to public *sinks*. In a web application, for example, IFC can prevent the flow of sensitive data (e.g., passwords, credit card information, etc.) to malicious domains by means of web APIs like the `XMLHttpRequest` API or setting the `src` property of images, which are sinks of information. An IFC policy requires sources to be labeled with an appropriate *security label*. For example, non-sensitive sources producing values that are considered to be publicly observable could be associated with a "low" label L. In contrast, sensitive sources producing values that should not be observable outside the application could be tagged with a "high" label H. By also labeling sinks, an IFC policy could then disallow H values from flowing to L sinks. Conceptually, IFC associates values with the same label as the source that produced them, and also lifts all operations on values to correctly label their output (e.g., the concatenation of a H string with a L string is a H string).

978-1-7281-0591-8/19/$31.00 © 2019 IEEE

1) IFC Enforcement: IFC policies can be verified ahead of time by means of static analysis or dynamically enforced through runtime monitoring. Due to its dynamic features (`eval`, dynamic function creation, etc.), a dynamic enforcement approach is often preferred for JavaScript and web applications [3]–[5]. Runtime monitoring can be achieved by modifying the browser's interpreter, by running an interpreter on top the browser interpreter, or by instrumenting the application's source code. Given the number of browser vendors and browser versions, code instrumentation is best suited.

2) Types of IFC: There are two types of information flow that determine how values and their security labels are disseminated in an application [1]. First, an *explicit* flow represents a direct propagation of values. For example, in the variable declaration statement `var y = x;` there is an explicit flow from `x` to `y`, and the value of `y` will have the same label as the value of `x`. Second, an *implicit* flow of information involves value flow that depends on control flow. For example, in the conditional statement `if (h) y = 1 else y = 0;` there is an implicit flow from `h` to `y`, and the value of `y` will have the same label as the value of `h`. Besides `if`, other control constructs that cause implicit flows are `while`, `break`, `continue`, `return`, and `throw`.

3) IFC Challenges: Handling implicit flows is already a challenging task for any dynamic IFC approach. Additionally, several features and characteristics of web applications complicate the process of tracking information flow by an IFC monitor. For example, the functionality of reacting to user input and user-generated events or performing network requests are implemented in *external libraries* such as the JavaScript Standard Library and the Document Object Model (DOM), and exposed by means of web APIs. The internals of these web APIs are not available (they are often implemented in other languages such as C++) and therefore have to be considered as black boxes by the IFC monitor. This makes it impossible for the IFC monitor to know how the information flows within an external library. As another example, *dynamic code evaluation* (e.g `eval`) allows the program to execute unknown code represented as a string value.

B. GIFC*: Practical IFC for Web Applications*

In prior work we developed GIFC [2], an IFC mechanism for JavaScript. GIFC uses code instrumentation to weave an IFC monitor into the application source code, making it portable across runtime engines and browser vendors. GIFC's monitor is based on the *permissive upgrade* technique [3], which makes the enforcement more precise. GIFC is also able to deal with dynamic code evaluation, and features a function model mechanism that enables information tracking through APIs calls and DOM.

GIFC's IFC policy specification interface is composed of two functions: `tagAsSource` and `tagAsSink`. `tagAsSource(exp)` allows developers to indicate that values produced by expression `exp` should be handled as sensitive. `tagAsSink(exp)` tags `exp` value as a sink of information. After the sources and sinks of the program are defined, GIFC's enforcement mechanism prevents any explicit or implicit flow of sensitive information to a public sink.

Listing 1. Specification of sources and sinks.

```
tagAsSink(console.log);                                        1
                                                               2
function getPassword() {                                       3
  var pass= document.querySelector("#pass").value;             4
  return tagAsSource(pass);                                    5
}                                                              6
                                                               7
function singleAssign() {                                      8
    var pass = getPassword();                                  9
    console.log(pass);                                         10
}                                                              11
```

1) Example: Listing 1 shows a concrete example how a developer can specify sources and sinks in GIFC. Line 1 tags the `console.log` function as a sink. The user-provided password in line 4 is considered sensitive and therefore tagged as such (line 5) before exiting the function. The function `singleAssign` attempts to log the password to the console (line 10), but this is captured and blocked by GIFC's enforcement mechanism.

III. Approach: Machine Learning-Assisted IFC

The manual annotation of all sources and sinks is a tedious task for programmers employing IFC policies. For this reason GUARDIAML is equipped with a machine learning (ML) component for automating parts of the process of identifying sources and sinks, alleviating the burden on developers. Given a function call, the ML component should predict whether this function call targets a source, a sink, or neither. The ML algorithm employs Support Vector Machines (SVMs) and we used an annotated Node.js API as training data. We opted for SVM as it is a very popular and successful classification method within the ML literature and it has been shown to work in a similar task before [6], making it a natural choice. For the ML component, a sink is defined as a call to a resource method that either creates or overwrites a previous value, and a source is defined as a call to a resource method that reads and returns a value at the application code.

We draw inspiration from [6], in which ML is used to classify sinks and sources in the Android API. This approach is, however, specific to how the Android API is coded and what conventions are used, which is reasonable as it is application specific. In this work, we focus on the Node.js API and devise an approach to deal with sinks and sources specific to Javascript by using the language's idioms.

1) Support Vector Machines: SVMs are AI technique which provides a discriminative, max margin classifier and is formally defined by separating hyperplanes. It is a suitable algorithm in our setting as it is a supervised method which finds the optimal hyperplanes to separate the categories in a given dataset. Specifically, it finds the hyperplane that minimizes the maximum distance to the decision boundary. Loosely, what the algorithm finds is the distance of the data points to a decision boundary, which has been calculated such that the maximum distance of data points is minimized. For an in-depth discussion on SVMs we refer the reader to [7].

2) Dataset Creation: In this work we use the Node.js API to train GUARDIAML's classifier. However, data regarding the Node.js API is not available in a usable form to apply supervised machine learning (i.e., there exist no labelled dataset). Transforming and enriching the API specification to make it a suitable target for supervised ML was a long and manual process. To encode the labelled dataset, we use a json-formatted text that includes the necessary information to apply ML. This is further explained in Section IV-D.

Our approach defines features of inputs that are fed to the SVM for training based on characteristics of naming, such as whether a method starts with a specific keyword such as `get` or `set`. We devised a list of common keywords for sinks and sources for Javascript as a driving force, as this would increase the applicability of our approach to other APIs written in that language. Of course, this design decision makes the algorithm sensitive to particular naming conventions. However, we argue that if coding conventions are followed, we expect our algorithm to generalize well to unseen instances of different APIs coded in Javascript. In case an API uses a very specific naming convention, then features need to be re-designed and a new SVM trained, which does not limit applicability of our approach as it is a straightforward process.

Annotating examples is important in order to use supervised methods to tackle this problem. We performed labelling on some of the methods of the API manually in order to create a training and test set. The details are the following:

- 265 labelled examples (out of 510)
- 21% are *sinks*
- 30% are *sources*
- 49% are *neither*

We applied machine learning on the labeled dataset in order to learn how to handle new unseen method specifications, or entire APIs, and automatically classify each method into one out of three possible classes: sink, source or neither.

3) Performance: The SVM was trained by first splitting the annotated data into a training set and a test set at 80% to 20% ratio. This is standard procedure in ML research to be able to evaluate an algorithm's performance on unseen data. The classifier is using the training data in order to learn the optimal way to correctly classify each example to its associated class. After the SVM has been trained the results on the testing set are reported, which basically provides an assessment of how good the algorithm is in classifying the data. A trained classifier that performs well can then be applied to new, unseen examples with the expectation of correct classification.

TABLE I
SVM CLASSIFICATION FOR SINKS AND SOURCES IN THE NODE.JS API.

Class	Precision	Recall
neither	0.93	0.97
source	1.00	0.91
sink	0.94	0.94
$avg/total$	0.95	0.95

Table I shows the results the SVM classifier obtained for sinks and sources in the Node.js API. We present our results using *precision* and *recall*. Precision is defined as the ratio of true positives over the sum of true and false positives, i.e. the ability of a classifier not to label negative examples as positive. Recall is the ratio of true positives over the sum of true positives and false negatives, i.e., the ability of a classifier to find all the positive examples.

The results in Table I show that using an SVM for IFC analysis in the Node.js API is very successful. We managed to obtain 95% performance both for precision and recall. This trained SVM can therefore be given unseen method specifications and full APIs and classify new examples correctly. It can be used as a module in GUARDIAML to automatically identify sinks and sources so that security policies can be deployed in relevant places.

IV. IMPLEMENTATION

We implemented our machine learning-assisted IFC approach in a tool called GUARDIAML. Figure 1 shows the tool overall architecture, which consists of 5 components.

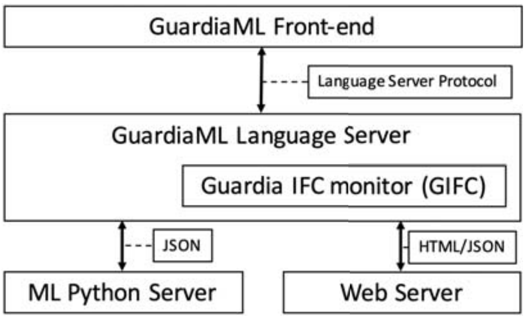


Fig. 1. GUARDIAML implementation architecture

The GUARDIAML *Front-end* is a VSCode plugin that offers commands to (1) ask for sources and sinks, or (2) instrument and execute a JavaScript application under IFC monitoring.

The GUARDIAML *Language Server* is a Node.js application that implements a Language Server Protocol (LSP) [8], which enables communication between development tools. Our GUARDIAML language server is responsible for processing all the commands and providing the diagnostics to the GUI.

The *Guardia IFC monitor (GIFC)* [2] is responsible for the source code instrumentation of either JavaScript programs or HTML pages. In the case of JavaScript source code, it is also responsible for executing it.

The *ML Python Server* is responsible for suggesting sources and sinks. It receives as input an array of JSON objects as shown in Figure 2. Each object encodes an expression in the *textRaw* property and its location in the *loc* property. The result will be the classification of each element as a *source*, *sink*, or neither.

The *Web Server* is responsible for serving instrumented HTML pages. When it is invoked, it first starts a web server and then opens a web browser with the URL pointing to the given input.

A. GUARDIAML *Components' Interaction*

Programmers interact with GUARDIAML through commands that in turn trigger the interaction between its components. In what follows, we detail how the components interact as a result of one concrete command, namely executing the GIFC monitoring mechanism.

Listing 2 shows how to register a handler that reacts to the *Execute IFC monitor* command (line 1). The handler receives the command's arguments (e.g., the editor source code) and calls the GIFC component. If there is an IFC policy violation during the execution of the code being analyzed, an error is thrown that includes the source code location of the violation. We use the term *diagnostic* to refer to a compiler error or warning that is shown in the user interface [9]. Line 8 sends the diagnostics to the GUARDIAML Front-end, which will update the GUI accordingly.

B. *Refactoring Driven by Machine Learning*

A *code action* [9] can be associated with a *diagnostic*. In GUARDIAML, code actions enables the refactoring of code to tag sources and sinks of information. The code actions and the refactoring code are computed in the *Language Server Component* as a result of the ML Python Server's suggestions.

Listing 2. Example of language server command handler.

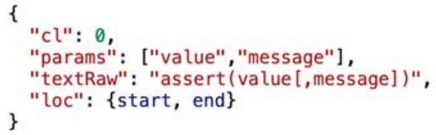
```
1  connection.onExecuteCommand((params) => {
2     let diagnostics = [];
3     try {
4        IFCMonitor.run(params.src);
5     } catch (error) {
6       diagnostics.push(IFCMonitor
7                  .diagnostics(error));
8       connection.sendDiagnostics({
9         uri: params.docUri.external,
10        diagnostics: diagnostics
11      });
12    }
13 })
```

C. *Scanning the Application*

In order to give suggestions of sources and sinks, the ML component is provided with every function call expression that appears in an application's source code. To obtain all function call expressions we build the *abstract syntax tree* (AST) of the source code using *Acorn.js* [1]. The resulting AST is then traversed using *Estraverse* [2], visiting all its nodes looking for *CallExpression* nodes. For every *CallExpression*, we collect the expression source code and its AST location information as shown in Figure 2.

D. *ML Component*

We used scikit-learn [10], a Python library for ML, to train the SVM model. We used grid search to optimize for kernel selection and also used 5-fold cross validation. We found that the best results were obtained by using a radial basis function kernel.



```
{
  "cl": 0,
  "params": ["value","message"],
  "textRaw": "assert(value[,message])",
  "loc": {start, end}
}
```

Fig. 2. Input format for SVM.

1) Training: The input to the SVM is given in the form of JSON following the specification shown in Figure 2. From this specification features are automatically extracted from the `textRaw` field. Each input's respective class is denoted in the `cl` field. The inputs come in the form of a binary representation based on the presence or absence of a feature. Table II shows a partial list of features used as input to the classifier assuming only 3 keywords; `get, set, log`; if a feature is present then its respective bit is set to 1.

TABLE II
SAMPLE FEATURES ASSUMING THREE KEYWORDS: GET, SET, LOG.

Function	Binary Representation
`getPassword(u)`	001
`setEmail(e)`	010
`console.log()`	100

2) Prediction: The server loads the trained SVM model and waits for a request for classification. The server expects a method, or lists of methods, in JSON format as shown in Figure 2. For prediction, only the `textRaw` is needed. The JSON file may contain more information, such as metadata about the location of a method in the source code for displaying diagnostics. Prediction does not alter any of provided input values but results in the addition of a new field, `cl` to indicate whether a method is a source (1), a sink (2), or neither (0).

V. GUARDIAML IN ACTION

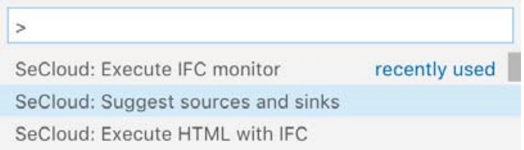
In this section we describe GUARDIAML from the programmer's perspective by means of an concrete example extracted from the IFC benchmarks used in prior and related work [2], [11]. The goal of this example, shown in Figure 4, is to leak the length of a user password `pass`, of which the input is simulated by function `getInput()`, in function `chkpassword()`. To this end, a **`for`** loop counts up from `0` to `16`, throwing an exception when loop counter variable `j` is equal to the password length. At line 13 in the **`catch`** block, the code attempts to log the value of the loop counter (now bound to `len`) that corresponds to the actual password length in case it is less than 16 characters long.

In order for developers to test whether their application is leaking information, they need to declare IFC policies by annotating sources and sinks of sensitive information. Instead of doing this manually, GUARDIAML allows developers to select the command *Suggest sources and sinks* from the command

[1] https://github.com/acornjs/acorn

[2] https://github.com/estools/estraverse

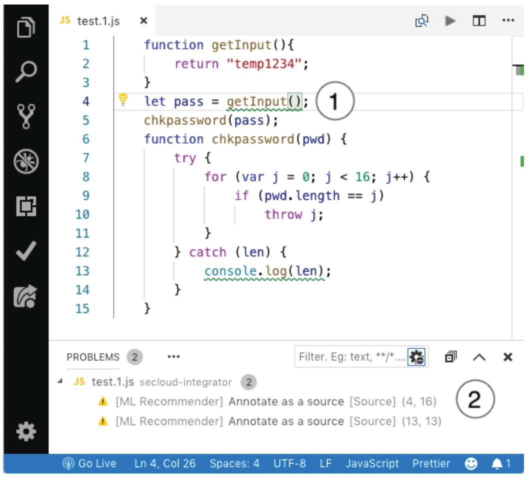
palette of its VSCode interface (Figure 3). Any suggestions are then highlighted in green in the source code and also are listed in the problems panel of the IDE (Figure 4). Clicking on a suggestion applies a code refactoring for that suggestion which makes GUARDIAML annotate the corresponding expression as a source or a sink (Figure 5). Once all sources and sinks are identified and annotated, the programmer can execute an instrumented version of the program by selecting the *Execute IFC monitor* command. The monitoring dynamically enforces IFC and any information flow violation will halt the execution. The source code location of where a violation occurred is reported as a diagnostic to GUARDIAML's Front-end. The diagnostic is shown in the problems panel and by highlighting the corresponding code in red (Figure 6).



SeCloud: Execute IFC monitor

SeCloud: Suggest sources and sinks
SeCloud: Execute HTML with IFC

Fig. 3. GUARDIAML commands.



```
function getInput(){
    return "temp1234";
}
let pass = getInput();
chkpassword(pass);
function chkpassword(pwd) {
    try {
        for (var j = 0; j < 16; j++) {
            if (pwd.length == j)
                throw j;
        }
    } catch (len) {
        console.log(len);
    }
}
```

PROBLEMS 2

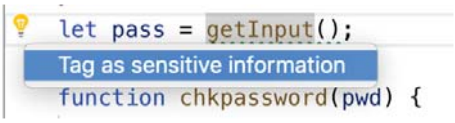
Filter. Eg: text, **/*....

JS test.1.js secloud-integrator 2

⚠ [ML Recommender] Annotate as a source [Source] (4, 16)

⚠ [ML Recommender] Annotate as a source [Source] (13, 13)

Fig. 4. Sources and sinks suggestions.



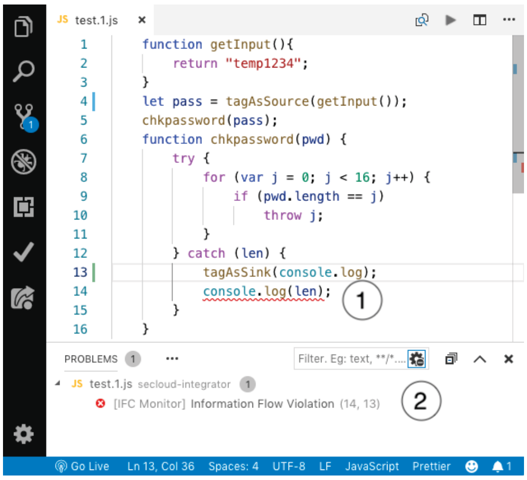
```
let pass = getInput();
function chkpassword(pwd) {
```

Fig. 5. Tag as source of sensitive information.

A video showing the features of GUARDIAML can be found at: https://youtu.be/LkwpeSJ9K5Q

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented GUARDIAML, a machine learning-assisted IFC plugin for Visual Studio Code. GUARDIAML adds IFC to the developer's toolbox, enabling the verification of the confidentiality of web applications. The suggestions of sources and sinks using ML alleviates developers of the burden of manually specifying sources and sinks. Moreover, the tool may suggest security-relevant points in an application of which developers may not be aware.



```
function getInput(){
    return "temp1234";
}
let pass = tagAsSource(getInput());
chkpassword(pass);
function chkpassword(pwd) {
    try {
        for (var j = 0; j < 16; j++) {
            if (pwd.length == j)
                throw j;
        }
    } catch (len) {
        tagAsSink(console.log);
        console.log(len);
    }
}
```

PROBLEMS 1

Filter. Eg: text, **/*....

JS test.1.js secloud-integrator 1

[IFC Monitor] Information Flow Violation (14, 13)

Fig. 6. Information flow violation example.

As future work, we plan to train the SVM model with other APIs such as the DOM to cover a wider range of information sources and sinks. It would also be interesting to augment GUARDIAML with functionality to facilitate the debugging of IFC violations by tracing back and visualizing information flows from a sink to its sources.

REFERENCES

[1] D. Hedin and A. Sabelfeld, "A Perspective on Information-Flow Control." *Software Safety and Security*, 2012.

[2] A. L. S. Pupo, L. Christophe, J. Nicolay, C. De Roover, and E. G. Boix, "Practical Information Flow Control for Web Applications." *RV*, vol. 11237, no. 5, pp. 372–388, 2018.

[3] T. H. Austin and C. Flanagan, "Permissive dynamic information flow analysis." *PLAS*, pp. 1–12, 2010.

[4] A. Bichhawat, V. Rajani, D. G. 0001, and C. H. 0001, "Generalizing Permissive-Upgrade in Dynamic Information Flow Analysis." *CoRR*, vol. cs.CR, pp. 15–24, 2015.

[5] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu, "A Survey of Dynamic Analysis and Test Generation for JavaScript," *ACM Comp. Surveys*, vol. 50, no. 5, pp. 1–36, Nov. 2017.

[6] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks." in *NDSS*, 2014.

[7] T. M. Mitchell, "Machine learning book," 1997.

[8] Language Server Protocol. [Online]. Available: https://microsoft.github.io/language-server-protocol/

[9] "Visual studio code api reference," Apr 2016. [Online]. Available: https://code.visualstudio.com/docs/extensionAPI/vscode-api

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[11] B. Sayed, I. Traoré, and A. Abdelhalim, "If-transpiler: Inlining of hybrid flow-sensitive security monitor for JavaScript," *Computers & Security*, vol. 75, pp. 92–117, Jun. 2018.