



# IBE.js: A Framework for Instrumenting Browser Extensions

Elvira Moreno-Sanchez<sup>1</sup><sup>a</sup> and Pablo Picazo-Sanchez<sup>2</sup><sup>b</sup>

<sup>1</sup>*IMDEA Software Institute, Madrid, Spain*

<sup>2</sup>*School of Information Technology, Halmstad University, Sweden*

**Keywords:** Browser Extensions, Quality, Testing.

**Abstract:** Millions of people use web browsers daily. Extensions can enhance their basic functions. As the use and development of browser extensions grow, ensuring adequate code coverage is essential for delivering high-quality, reliable, and secure software. This paper introduces IBE.js, a framework to monitor and assess the coverage of browser extensions. IBE.js conducts an analysis of the main JavaScript files, background pages and content scripts, of 4,495 browser extensions from the Chrome Web Store. By utilizing a blank HTML file, we found that on average, more than 33% of the lines in these scripts are executed automatically. This coverage represents the number of lines executed by default, without any influence from user interaction or web content. Notably, IBE.js is a versatile framework that can be utilized across various platforms, ensuring compatibility with extensions from other web stores such as Firefox, Opera, and Microsoft. This enables comprehensive coverage analysis and monitoring of extensions beyond a single browser ecosystem.

## 1 INTRODUCTION

Due to their advantages, browsers have become ubiquitous tools on almost all computers. According to statistics, as of May 2023, Chrome is the most widely used browser with 65.74% market share, followed by Safari with 18.86%, Microsoft Edge with 4.27%, Firefox with 2.92%, and Opera with 2.27% (Oberlo, 2023). Specifically, Chrome has over 900 million users.


Google Chrome browser offers the popular Web Store (Google, 2023b), an online marketplace operated and maintained by Google. The software stored in the Chrome Web Store belongs to one of the eleven categories: Accessibility, Blogging, Communication, Fun, News, Shopping, Photos, Productivity, Search Tools, Sports, and Web Development. In particular, in the Web Store we can find extensions and visual themes for the browser as well as web applications.


Browser extensions are small pieces of software that most browsers nowadays allow users to install to enrich their user experience. Changing the screen background to help color-blind people, extracting the URLs of every webpage the user visits, or blocking advertisement pop-ups automatically are only a few examples of the functionality provided by browser ex-

tensions. One of the main reasons for the rise of extensions is that they are based on web languages such as JavaScript, CSS, and HTML, which makes it easy for any developer to create their extensions.

**Software Analysis.** To analyze extensions or any application in general, there are two main strategies: static and dynamic analysis (Ernst, 2003). While the former is faster, it is impossible to statically determine whether a line of code will be executed (reachable) or not at run-time and, although it can be useful to minimize the number of tests to be performed, it is not a substitute for such tests. Dynamic analysis was introduced to solve this problem. However, despite its advantages, such as testing, performance monitoring or debugging, dynamic analysis is computationally demanding and time-consuming.

In the case of JavaScript, an untyped scripting language, even using both techniques, if a line of code has not been executed, we cannot claim that it will not be reachable in the future. Analyzing the lines of code of extensions and executing them all to study their conditions is nearly impossible. To illustrate this problem, imagine for example that `chrome.cookies()` is only executed when a specific element (`<div id=""></div>`) whose id is generated at runtime consisting of the user's location and date.

<sup>a</sup> <https://orcid.org/0000-0001-8551-6572>

<sup>b</sup> <https://orcid.org/0000-0002-0303-3858>

**Code Coverage.** One challenge in software testing is evaluating the quality of test cases since the higher the quality, the more bugs are found. This is known as *test adequacy criteria* and the most used metric to measure such a criterion is the *code coverage* (Zhu et al., 1997). Code coverage is a software test that improves the allocation of testing resources. It is defined as an indicator of the effectiveness and completeness of testing to select and evaluate test cases. Also, it has been proved that high code coverage means fewer bugs and higher software availability (Rapps and Weyuker, 1985).

In JavaScript, code coverage is crucial since: 1) JavaScript is often used to add dynamic functionality to web pages and code coverage might help to ensure that the code does not contain security vulnerabilities (e.g., Cross-Site Scripting (XSS)) which can allow attackers to inject malicious scripts into web pages (Melicher et al., 2018); 2) it helps to ensure that sensitive information is handled securely, protecting it from unauthorized access or manipulation. It might also help to ensure that this data is collected and processed in a way that protects user privacy and complies with privacy regulations (Ou et al., 2022); 3) it might verify security controls that JavaScript implements, such as authentication and access controls (Sun et al., 2011).

Code coverage encompasses a variety of types (Marick et al., 1999), including i) function coverage, which measures whether or not a function is called; ii) branch coverage, which measures whether or not a branch of code is executed; iii) statement coverage, which measures whether or not a statement is run (but does not provide information on how often it is executed), and; iv) line coverage, which measures the exact number of times a line of code is executed.

**Contributions.** In this work we propose IBE.js, a simple but powerful solution to instrument the source code of browser extensions to obtain their coverage. Specifically, we focus on obtaining what we call “*Ground truth coverage*”, that is, the coverage of the extensions when visiting a blank HTML page. No element such as HTML content, user interaction or JavaScript events might alter the coverage of the extensions code. In short, the ground truth coverage lets us know what the lines of source code shipped by extensions executed by default are, i.e., regardless of Web content, JavaScript events, and without any user interaction.

To illustrate some of the advantages of IBE.js, suppose we have an extension that changes the background color of the HTML to a black one whenever there is a video in the content. Such an extension has

a ground truth coverage of 10%, i.e., when executing the extension using a blank webpage. We can then examine how the web content influences an increase in coverage. As a consequence, we can establish a connection between specific functionalities of the extensions, triggered by a particular webpage, and the resulting coverage expansion. Additionally, we can conduct similar analyses involving user interaction, as well as a combination of both user interaction and web content. With IBE.js, we can certify that the coverage of the extension increases to 80% when accessing to any webpage with videos such as Youtube, while remaining at 10% on webpages without videos. Note that, in an hypothetical scenario where the coverage of the extension unexpectedly increases when the user accesses her bank account without any video in the HTML, IBE.js can detect such changes and potentially raise security concerns.

In detail, our contributions are:

- We propose IBE.js, a framework that uses a combination of static and dynamic analysis to instrument the source code of browser extensions and execute them to get the code coverage (see Section 3). We publicly release our framework for future research on this area<sup>1</sup>.
- We got the ground truth coverage of the extensions and conclude that, on average, 33.66% of the background pages and content scripts’ source code is executed (see Section 4).
- We perform a second analysis called validation coverage to validate the results got in the ground truth coverage (see Section 4.1).

The rest of the document is structured as follows: in Section 2, we explain some basic concepts to understand the paper; in Section 5, we present the related work, and; finally, in Section 6, we show the conclusions drawn from this research.

## 2 BACKGROUND

In this section, we introduce some concepts and definitions we use in the paper.

### 2.1 Browser Extensions

Browser extensions consist of a mandatory file called *manifest.json* and as many optional static files as needed. The manifest file holds basic metadata, like the extension’s name, permissions, and version (Google, 2021). It sets the limits on API calls the extension can access and defines its scope. Among

<sup>1</sup><https://github.com/elviraimdea/IBE.js.git>

the static files, background and content scripts are the most significant as they determine the extension's logic (Google, 2023a).

Background pages are scripts loaded simultaneously as the extension and remain so until the extension itself is uninstalled, or disabled (Google, 2023a). Background pages run in the browser and can access different information as they have full access to the extension's API once the necessary permissions are declared (Chen and Kapravelos, 2018).

Content scripts run within web content and lack access to most of the browser's privileged APIs. They can be injected into web pages through the manifest file or dynamically by the background page.

## 2.2 Abstract Syntax Tree

There are numerous tools for static code analysis (Sonarqube, 2023; SonarCloud, 2023; Spectral, 2023; Kundel, 2020). In particular, an Abstract Syntax Tree (AST) (Kundel, 2020) is a tree-like representation of the syntax of a source code program according to the formal grammar of a programming language that describes the syntactic structure of the program. IBE.js uses ASTs for instrumenting browser extensions. In the following, we detail the main advantages of using ASTs.

*Lexical analysis*, also known as tokenization, consists of converting each element of the source code into tokens. These tokens are used to identify the type and value of the piece of code to which they refer. In Listing 1, we show an example of lexical analysis of “var aux = "Hello World"”, where we see that the lexical analysis is only separating the sentence by type and value.

```
[{ type: "VariableDeclaration", value:
  "aux"},
 { type: "String", value: "Hello World
"}]
```

Listing 1: Lexical analysis AST of “var aux=Hello World”.

*Parsing*, also known as syntax analysis, is the step that converts the tokens generated in the lexical analysis into an AST. This parsing is the one that gives meaning to the code as it represents its structure, i.e., we can know the function that the code performs.

*Code generation* allows developers to safely manipulate or even change the whole structure of an AST.

## 2.3 ECMAScript

ECMAScript (ECMA, 2021), is a language specification mainly used in general-purpose programming

languages, being JavaScript the most known one. Even though ECMAScript was initially designed for client applications, with the incipient popularity of JavaScript, it also allows coding server-based software. Therefore, ECMAScript provides not only client-side computation, including objects representing menus, windows, and text areas but also server-side functionality like objects representing requests, clients, and file system management (ECMA, 2021). Each browser and web server that supports ECMAScript provides its own host environment, completing the ECMAScript execution environment.

In this paper, we use Esprima<sup>2</sup> for the generation of ASTs. Esprima is a high-performance, standards-compliant ECMAScript parser written in ECMAScript.

## 2.4 Parameters for Statistical Study

We define the main parameters for the statistical study we carry out to obtain the code coverage. These are: confident level, confident interval and sample size (Israel, 1992).

The *confidence level* gives the degree of assurance we can have. This parameter is expressed as a percentage so that a confidence level of 100% indicates that it is certain that the results will not change when the experiment is repeated. On the other hand, 0% indicates that there is no chance that repeating the experiment will produce the same results.

The *confidence interval* is, in short, the statistical error that arises when the sample does not represent the entire population. For example, if we use a confidence interval of 3 and 50% of the population choose an answer, we can be confident that if we ask the question again, the entire relevant population would be between 47% and 53%.

*Population size* is known as the total number of elements in a study. On the other hand, sample size refers to the number of observations included in a study. Sample size has an impact on two statistical properties: 1) the precision of the estimates and; 2) the study's ability to draw conclusions.

## 3 IBE.js: INSTRUMENTING BROWSER EXTENSIONS

IBE.js is a framework that uses a combination of static and dynamic analysis to instrument the lines of code of the JavaScript files of extensions and thus, gets the coverage. This value is calculated as the fraction of

<sup>2</sup><https://esprima.org>

lines of code executed in a program at least once. Note that comments and blank lines are not counted as lines of code. The code coverage allows us to know how many lines of code are executed by default in the background pages and content script files of the extensions, i.e., regardless of web content and without any user interaction. Also, it is worth mentioning that IBE.js, does not need to implement any complex architecture based on honeypages (Kaparavelos et al., 2014; Solomos et al., 2022) nor visit any of the webpages (Sjosten et al., 2019) the extensions are suppose to work to analyze and capture the coverage.

### 3.1 IBE.js Architecture

Figure 1 shows IBE.js’s architecture. The left part represents the static analysis, where IBE.js parses the manifest file and instruments the extensions. The right part of the figure covers dynamic analysis, which includes automatically installing the instrumented extensions on the web browser and calculating code coverage.

#### 3.1.1 Static Analysis

First, IBE.js parses the manifest file of the extensions and extracts all the JavaScript files defined in the "content\_script" and "background" keys. Second, IBE.js skips scripts that belong to any well-known library by computing the Subresource Integrity (SRI) and comparing such a value with the ones provided by some of the most popular Content Delivery Networks (CDNs) like Google<sup>3</sup> and cdnjs<sup>4</sup>.

The instrumentation of the code takes place in the static analysis module. The code instrumentation is the method of adding "fetch()" commands between the AST nodes of each file. Such a command is a JavaScript function that makes HTTP requests to a server. IBE.js requires that these commands follow the following format: `fetch("<URL>/<extension_id>/<file_path>/line/<line_number>")`. These parameters include: 1) the URL of a Web server; 2) the ID with which the extension is identified in the Web Store; 3) the relative file path to be instrumented in the extension, and; 4) the number that corresponds to the instrumented line of the file.

Let us illustrate the instrumentation process with a random file called engine.js from a random extension whose ID is "acaamclplaocnfddlcllkbeaelpipgkm". Concretely, we chose an inline conditional statement, very common in programming languages such as Python, Ruby or JavaScript (see Listing 2).

<sup>3</sup><https://developers.google.com/speed/libraries>

<sup>4</sup><https://cdnjs.com>

```
(function (t) {t.enabled ? r() : e.w("
  app is disabled, do nothing to
  taobao page")})
```

Listing 2: Original code from engine.js file.

After IBE.js instruments the code, it is divided into a series of if-else statements while preserving the original code’s conditions and alternatives.

```
fetch('http://localhost/
  acaamclplaocnfddlcllkbeaelpipgkm/
  javascript/taobao/engine.js/line/0'
);
(function (t) {
  fetch('http://localhost/
    acaamclplaocnfddlcllkbeaelpipgkm/
    javascript/taobao/engine.js/line
    /1');
  if (t.enabled) {
    fetch('http://localhost/
      acaamclplaocnfddlcllkbeaelpipgkm
      /javascript/taobao/engine.js/
      line/2');
    r();
    fetch('http://localhost/
      acaamclplaocnfddlcllkbeaelpipgkm
      /javascript/taobao/engine.js/
      line/3');
  } else {
    fetch('http://localhost/
      acaamclplaocnfddlcllkbeaelpipgkm
      /javascript/taobao/engine.js/
      line/4');
    e.w('app is disabled, do nothing
      to taobao page');
    fetch('http://localhost/
      acaamclplaocnfddlcllkbeaelpipgkm
      /javascript/taobao/engine.js/
      line/5');
  };
  fetch('http://localhost/
    acaamclplaocnfddlcllkbeaelpipgkm/
    javascript/taobao/engine.js/line
    /6');
});
fetch('http://localhost/
  acaamclplaocnfddlcllkbeaelpipgkm/
  javascript/taobao/engine.js/line/7'
);
```

Listing 3: Instrumented code from engine.js file.

Note that adding inline "fetch()" directly into background pages and content scripts might be in conflict with both permissions and Content Security Policy (CSP) of the extensions. To bypass such limitation, IBE.js automatically parses the manifest file

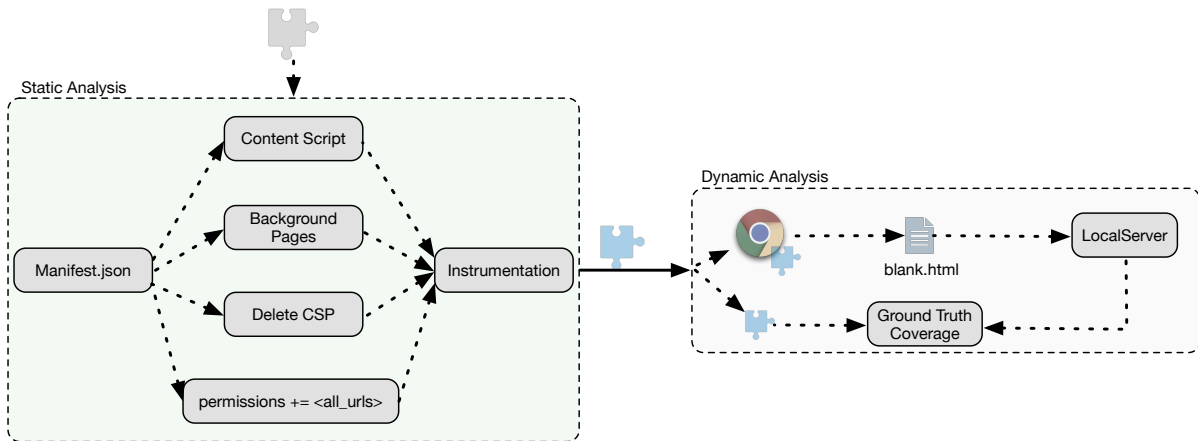


Figure 1: IBE.js system architecture.

and: 1) deletes the CSP property of the manifest file in case extensions define it, and; 2) adds the `<all_urls>` string to the permissions in case it is not already included. Performing these changes in the manifest file do not alter any of the functionality of the extensions. Rather, we allow them to work in our controlled server as if they were executed in a real scenario, i.e., in the webpages they were coded to work in.

The instrumentation results in the number of instrumented lines, i.e., the number of added fetch statements to the original file. If the files can be instrumented without error, the original scripts are replaced with the same file that includes the added lines. However, if any errors occur during the process, the original scripts remains and the total number of instrumented lines is “-1”.

### 3.1.2 Dynamic Analysis

The objective of the dynamic analysis module in IBE.js is the installation and execution of the extensions. To do so, IBE.js relies on a combination of Python and Puppeteer<sup>5</sup> to automatically launch the browser and install the corresponding extensions. Later, every extension executes the “`fetch()`” to the server specified in the command, a server that IBE.js automatically sets up every time a new extension is installed and executed.

Once all the requests have been made, IBE.js automatically stops the server and obtains all the data necessary to get the code coverage. In more detail, we: 1) calculate the coverage of each file (see Equation (1)); 2) group the files according to the extension they belong to and calculate an average of the coverage of those files; 3) group the extensions by the category they belong to, and get the average of the

coverage of those extensions, and; 4) obtain an average of the total coverage of all the categories.

$$coverage(\%) = \frac{coveredLines}{totalLines} * 100 \quad (1)$$

Let us illustrate the entire process with the same extension as in the previous example whose ID is “acaamclpaocnfddllcllkbealpipgkm”. Using Equation (1) and the data corresponding to the columns “Lines” and “Covered Lines” (see Table 1) we get the coverage of each file of the extension. For example, in the file named “engine.js”, the coverage is as follows:

$$coverage(\%) = \frac{17}{19} * 100 = 89.47$$

After this, we calculate the average coverage of the files of the same extension, obtaining an average coverage of 48.75%.

Table 1: Code coverage of acaamclpaocnfddllcllkbealpipgkm extension version 1.0.1.2.

Filename	Lines	Covered Lines	Files Coverage
debug.js	7	7	100%
engine-product.js	20	9	45%
engine.js	19	17	89.47%
pattern.js	6	6	100%
setting-dialog.js	41	0	0%
taobao-util.js	23	6	26.09%
util.js	68	0	0%
modaldialog.js	34	10	29.41%

## 3.2 Scope and Limitations of IBE.js

During the instrumentation, we found some code cases that limited the number of lines that could be instrumented. In the following, we explain the instrumentation process of some of them as well as some of the limitations IBE.js has.

<sup>5</sup><https://github.com/puppeteer/puppeteer>



**Manifest File.** IBE.js traverses the manifest file of all extensions to extract the JavaScript files to be instrumented. Therefore, these files belong to the "content\_script" and "background" tags of the manifest file. However, extensions can define HTML files as background pages and include external JavaScripts in them. These files are not analyzed by IBE.js. We are working on an extension that parses all the HTML and instruments the JavaScripts defined in the HTMLs.

**Obfuscated Code.** Google detected that over 70% of malicious code in the Web Store contained some sort of obfuscated code (Wagner, 2018). On October 2018, Google no longer accepted extensions with obfuscated code, and they even removed from the Web Store those extensions shipping such files.

**Adding Brackets.** In JavaScript, as in many other languages, it is not mandatory to add special characters to define a code block. For instance, JavaScript allows developers to not include curly brackets in some statements like `if`, `else`, `for`, `forin`. This is a problem because when adding the "`fetch()`" commands after each node, it will return an error because the `if` statement will not be well defined. We addressed this by automatically adding a **BlockStatement**, preserving the code's functionality, allowing us to instrument it and obtain coverage.

**Inline Statements.** JavaScript, like Python and PHP, allows inline statement definitions. One example is the `if..else` statement, which can be expressed as `<test>? <consequent> : <alternate>;`. We expanded such inline statements and created extended `if..else` blocks with the original `<test>` in the condition, and generated **BlockStatement** for both the `<consequent>`, and the `<alternate>`.

**Variable Declaration.** Even though our instrumentation method is sound, it is not complete. One such example of statement that we did not instrument is the variable declaration based on a conditional inline statement, e.g., `var <variable> = <test> ? <valueIf> : <valueElse>;`. We are working on a solution that generates an extended version of a **ConditionalStatement** (`if..else`) to instrument it as well as declaring the variable (`<variable>`) on every path of the **ConditionalStatement** path with its corresponding value (either `<valueIf>` or `<valueElse>`).

**Esprima.** We use Esprima to perform lexical analysis. However, some files are not compatible with this library and therefore cannot be parsed. For example, the snippet shown in Listing 4 throws an error in Esprima because it needs the `catch` statement to contain a parameter, e.g., `catch (error){alert("Error");}`.

```
try{$("#btn_capture").hide();}
catch{alert("Error");}
```

Listing 4: Esprima error.

We could have used other more error-tolerant JavaScript parsers like `acorn-loose`<sup>6</sup>. However, in doing so, we might modify the source code of the extensions and alter the intended behavior. We restrict ourselves to the source code, no matter whether it contains errors or not.

**Enrichment of the Fetch Command.** As a proof of concept, we only send information about the line that has been executed (see Listing 3). However, since we are sending the information to a local server, in this command we can include more detailed information such as the state of the variables and more complex syntax such as recursion and closure. In addition, we can use the content of the log file generated in the dynamic execution to recreate the execution flow.

## 4 PROOF-OF-CONCEPT: GROUND TRUTH COVERAGE

We implemented a proof-of-concept of IBE.js in Python and deployed it on a Linux computer with Intel(R) Core(TM) i7-4790 CPU @3.60GHz, 16GB of RAM. Performing the entire process with the 3,807 extensions took 19 hours, 34 minutes, and 41 seconds, i.e., 13 seconds per extension on average.

**Dataset.** We crawled the Web Store as of February 2023 and downloaded 114,840 extensions, and got the category they belong in the Web Store (see the 2<sup>nd</sup> column of Table 2). To get a representative sample, we used "Sample Size Calculator"<sup>7</sup> with a confidence interval of 5% and a confidence level of 95%. This implies that if we repeat the experiment using the same parameters, the results will be within 5% of the previous ones 95% of the time. The third column of Table 2 shows the number of random extensions IBE.js analyzes per category.

<sup>6</sup><https://github.com/acornjs/acorn>

<sup>7</sup><https://www.surveysystem.com/sscalc.htm>

Table 2: Code coverage obtained by category.

Category	Extensions		
	Total	Analyzed	Coverage
Accessibility	10,786	371	37.67%
Blogging	1,858	318	34.51%
Communication	10,547	371	32.60%
Fun	14,495	374	38.73%
News	2,278	329	36.07%
Photos	1,223	293	35.83%
Productivity	47,969	381	39.38%
Search Tools	5,787	360	18.76%
Shopping	6,341	362	37.47%
Sports	963	275	40.02%
Web Development	12,503	373	19.20%
TOTAL	114,840	3,807	33.66%

Once IBE.js analyzed the extensions randomly chosen for the experiment, we obtained the average coverage of each category (see Table 2). We can see, for example, that extensions in the Sports category have the highest code coverage (40.02%), i.e., extensions belonging to Sports execute 40.02% of the source code automatically regardless of the web content and the user interaction. On the contrary, extensions in the Search Tools category have the lowest code coverage (18.76%), i.e., extensions within the Search Tools category are highly dependent on either web content or user interaction.

With the coverage data obtained for each category, we have calculated that, on average, the coverage of the extensions analyzed is 33.66%. This means that in most cases, more than 50% of the lines that we instrumented are not covered when the dynamic analysis is performed in our controlled environment.

Table 3: Files coverage obtained by category.

Category	Extensions		
	Files	Covered Files	Files Coverage
Accessibility	2.86	2.69	94.63%
Blogging	1.72	1.66	97.10%
Communication	2.06	1.97	95.64%
Fun	1.88	1.83	97.21%
News	1.60	1.52	96.66%
Photos	1.86	1.75	93.50%
Productivity	2.90	2.77	93.88%
Search Tools	2.81	2.68	96.33%
Shopping	2.61	2.50	95.50%
Sports	1.95	1.85	94.93%
Web Development	2.38	2.21	94.51%

In Table 3 we measure the quality of IBE.js. We can see for each category, the average number of files of the extensions (2<sup>nd</sup> column), the average number of files successfully instrumented (3<sup>rd</sup> column), and the files coverage (4<sup>th</sup> column). We observe, that i) in general, the number of files that cannot be instrumented is very small, and; ii) the overall average

Table 4: Coverage of extensions belonging to Sport category.

	Extensions		
	Total	Executed	Coverage
Ground Truth	963	275	40.02%
Validation	963	963	41.73%

file coverage is of 95.38%. With this, we show that, in most cases, the majority of files of each extension are instrumented.

#### 4.1 Threats to Validity

To validate the results obtained when computing the ground truth coverage, we carried out a second experiment which we call *validation coverage* where we calculate the coverage of all the extensions of a single category.

We chose the Sport category, the one with the highest coverage and 963 extensions. This experiment gives us a result with a confidence level of 100% and a confidence interval of 0%, i.e., whenever we repeat the experiment, we will get 100% similarity in the results. The whole process, i.e., instrumenting, installing, and parsing the results, took us over 10 hours.

As shown in Table 4, we compared the results of the code coverage calculation for the Sports category in both experiments and found that the results are very similar, with 40.02% for the first execution and 41.73% for the second.

From the ground truth and validation coverage, we conclude that the results do not deviate from the confidence interval and that the coverage obtained for validation coverage is slightly higher. This increase means that, despite having increased the number of extensions to be analyzed (963) and the probability of finding files not covered is greater, we have obtained better results.

In Table 5, we show the total number of covered files for each experiment. The coverage of covered files is higher in the ground truth coverage than in the validation coverage (94.93% in the ground truth coverage and 94.91% in validation coverage). These results imply that, as the number of extensions to be executed increases, the number of erroneous files found also increases. However, such variation is not significant enough, so we can once again verify that the results obtained for the rest of the categories in ground truth coverage are perfectly valid.

#### 4.2 Cross-Platform Compatibility

We tested IBE.js with extensions stored in Google Web Store. This means that in Chromium-based

Table 5: Instrumented files of extensions belonging to Sport category.

	Extensions		
	Files	Covered Files	Coverage
Ground Truth	1.95	1.85	94.93%
Validation	1.96	1.86	94.91%

browsers, like Microsoft Edge, Opera and Brave, although the approach to extension distribution may vary between them (such as the use of Chrome Web Store), all extensions in these browsers are compatible with IBE.js. In practice, we used as input extensions from other web stores like Firefox Browser Add-ons<sup>8</sup>, Opera Add-ons<sup>9</sup> and Microsoft Edge Add-On<sup>10</sup> and certified that IBE.js successfully instrumented them.

Firefox is a non Chromium-based web browser developed by Mozilla that also accepts browser extensions (named add-ons). Although its architecture may differ in some cases, it maintains the same structure, using a directory containing a manifest file describing the background pages, content scripts and permissions, and the rest of the necessary files. We checked IBE.js using Firefox add-ons as input and obtained successful results, concluding that IBE.js also complies with Firefox add-ons.

## 5 RELATED WORK

In the following, we summarize the related work focusing when possible on JavaScript and more concretely, on browser extensions.

**Code Coverage.** J-Force (Kim et al., 2017) is a JavaScript-based application that scans multiple execution paths of browser extensions and detects malicious behavior by combining dynamic and static analysis. Authors experimented on 100 real-world JavaScript samples, and, as a result, they cover 95% of the code. A year later, in 2018, Hu et al. proposed JSForce, a framework that increases the code coverage in JavaScript to detect malicious scripts (Hu et al., 2018). JSCover (JSCover, 2023) is a specific tool for JavaScript coverage inspired in the popular JSCoverage<sup>11</sup>, a tool that measures code coverage for JavaScript programs. Other tools like BullseyeCoverage (BullseyeCoverage, 2023), Clover (Clover, 2023) and SLIMIUM (Qian et al., 2020) are not comparable

<sup>8</sup><https://addons.mozilla.org/en-US/firefox/>

<sup>9</sup><https://addons.opera.com/en/extensions/>

<sup>10</sup><https://microsoftedge.microsoft.com/addons/Microsoft-Edge-Extensions-Home>

<sup>11</sup><http://siliconforks.com/jscoverage/>

to ours as they focus on languages like C++, Java and browser code debloating respectively, so they cannot be used to measure the coverage of browser extensions.

However, of those tools that focus on JavaScript language, they do not take web content into account, unlike IBE.js, which does. This means that although we can manually select some of the content scripts from the extensions to be analyzed, this is far from the reality as content scripts can send messages to background pages. Recall that background pages are scripts that run in the background context of the browser, so they cannot be analyzed by such a tool. IBE.js offers a unique approach to measuring code coverage in browser extensions, taking into account web content and providing a more accurate assessment. Unfortunately, the closest tool to IBE.js, J-Force, is not online, and although we contacted the authors, we could not get the source code to execute and compare it.

**Code Analysis.** In browser extensions, although there are some authors who analyze extensions by grouping them into Aspects of Extension Behavior (AEBs) (Zhao and Liu, 2013), we can clearly identify three main approaches to analyze both their source code and behavior. The first strategy is statically analyzing the source code the extensions ship (Landi, 1992; Emanuelsson and Nilsson, 2008; Li et al., 2017; Fass et al., 2021). The second one is dynamically analyzing the behavior in a controlled environment (Ball, 1999; Kapravelos et al., 2014; Yerima et al., 2019; Chen and Kapravelos, 2018; Pilgun et al., 2018). Nowadays, most authors rely on a combination of both static and dynamic analysis (Wang et al., 2018; Pan and Mao, 2017; Eriksson et al., 2022; Starov and Nikiforakis, 2017). A very remarkable tool for its high accuracy in code analysis is VEX (Bandhakavi et al., 2010), which detects patterns and illuminates possible security vulnerabilities in Firefox extensions by classifying them as explicit flows. However, IBE.js differs significantly in that it uses a combination of static and dynamic analysis, instrumenting both the background pages and content scripts of extensions in web content.

**Code Instrumentation.** Yu et al. presented a method for implementing JavaScript code for browser security (Yu et al., 2007). Authors model a proprietary subset of JavaScript-CoreScript where they focus on the higher-order script and solve the problem of identifying and rewriting these scripts by infecting callbacks. Authors analyze 178,893 Chrome browser extensions using Mystique (Chen and Kapravelos,



2018), an information flow tracking tool based on taint analysis for browser extensions. FPTracker (Ashouri, 2019) is a standalone, portable and practical browser developed as a standalone tool. It combines static and runtime analysis of websites to track scripts for footprinting through integration with users. As a result, FPTracker analyzes the true purposes of encrypted scripts thanks to JavaScript instrumentation. However, these specific tools focused on browser extensions, such as Mystique, mainly focus on extension tracking based on taint analysis techniques, which are far from our goal. Hulk (Kaprauelos et al., 2014) dynamically install extensions and using a combination of honeypages and event-driven approach, authors try to execute as much functionality as possible. Note that this approach differs from ours, where we first statically instrument extensions and later we dynamically install them to check their code-coverage.

In conclusion, IBE.js differs from other tools in its focus on analyzing the JavaScript files of browser extensions. To achieve this, IBE.js uses a combination of static and dynamic analysis to instrument the extension files and obtain the final coverage. In addition, IBE.js takes web content into account, allowing it to provide a more complete and accurate analysis of extension coverage. Overall, IBE.js presents itself as a simple and effective solution for monitoring and evaluating the code coverage of browser extensions.

## 6 CONCLUSIONS

In this paper, we presented IBE.js, a simple yet powerful solution to instrument the source code of the browser extensions and get their coverage.

We established a ground truth dataset by using a blank HTML, and found that IBE.js successfully instruments the background pages and content scripts of extensions with an overall average coverage of 95.38%. We validated our experiments, revealing that we instrument 94.93% of the background pages and content scripts of the extensions from Sports category on average and found that over 40% of their code is automatically executed. To further confirm the reliability of IBE.js, we repeated the experiment using all the extensions from the Sports category and found that we instrumented 94.91% of their scripts and determined that they automatically execute 41.73% of their code. This demonstrates that the sample size used to compute the ground truth coverage is sufficiently representative, even with variations in the number of used extensions.

To validate the compatibility of IBE.js with other browsers, we used as input extensions from other web

stores like Firefox, Opera and Microsoft Stores and certified that IBE.js successfully instrumented them.

We conclude that the coverage that marks the baseline of IBE.js has a value of  $29.39\% \pm 10.63$ , being this the minimum value from which it will be possible to increase the scope of lines executed by means of methods such as the triggering of JavaScript events.

## ACKNOWLEDGEMENTS

This project has received funding from the Spanish Ministry of Science, Innovation, and University under the project TED2021-132464B-I00 PRODIGY, from the Spanish Ministry of Science, Innovation, and University under the Ramón y Cajal grant RYC2021-032614-I, and from a gift by Intel Corporation.

## REFERENCES

- Ashouri, M. (2019). A large-scale analysis of browser fingerprinting via chrome instrumentation. In *ICIMP*.
- Ball, T. (1999). The concept of dynamic analysis. In *ES-EC/FSE*, pages 216–234.
- Bandhakavi, S., King, S. T., Madhusudan, P., and Winslett, M. (2010). VEX: Vetting Browser Extensions for Security Vulnerabilities. In *USENIX*.
- BullseyeCoverage (2023). Quickly find untested c++ code and measure testing completeness. <https://www.bullseye.com>.
- Chen, Q. and Kaprauelos, A. (2018). Mystique: Uncovering information leakage from browser extensions. In *CCS*, pages 1687–1700.
- Clover, O. (2023). OpenClover. <https://openclover.org>.
- ECMA (2021). EcmaScript. <https://262.ecma-international.org/12.0/>.
- Emanuelsson, P. and Nilsson, U. (2008). A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21.
- Eriksson, B., Picazo-Sanchez, P., and Sabelfeld, A. (2022). Hardening the security analysis of browser extensions. In *SAC*, pages 1694–1703.
- Ernst, M. D. (2003). Static and dynamic analysis: Synergy and duality.
- Fass, A., Somé, D. F., Backes, M., and Stock, B. (2021). Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *CCS*, page 1789–1804.
- Google (2021). Manifest file. <https://developer.chrome.com/docs/extensions/mv3/intro/mv3-migration/>.
- Google (2023a). Browser extension architecture. <https://developer.chrome.com/docs/extensions/mv3/architecture-overview/>.
- Google (2023b). Chrome web store. <https://chrome.google.com/webstore>.

- Hu, X., Cheng, Y., Duan, Y., Henderson, A., and Yin, H. (2018). Jsforce: A forced execution engine for malicious javascript detection. In Lin, X., Ghorbani, A., Ren, K., Zhu, S., and Zhang, A., editors, *SecureComm*, pages 704–720.
- Israel, G. D. (1992). *Determining sample size*, volume 25. University of Florida Cooperative Extension Service, Institute of Food and Agriculture Sciences, EDIS.
- JSCover (2023). Jscover. <https://github.com/tntim96/JS-Cover>.
- Kapravelos, A., Grier, C., Chachra, N., Kruegel, C., Vigna, G., and Paxson, V. (2014). Hulk: Eliciting malicious behavior in browser extensions. In *USENIX*, pages 641–654.
- Kim, K., Kim, I. L., Kim, C. H., Kwon, Y., Zheng, Y., Zhang, X., and Xu, D. (2017). J-force: Forced execution on javascript. In *The Web Conf*, pages 897–906.
- Kundel, D. (2020). Abstract syntax tree. <https://www.twilio.com/blog/abstract-syntax-trees>.
- Landi, W. (1992). Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337.
- Li, L., Bissyandé, T. F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., and Traon, L. (2017). Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67–95.
- Marick, B., Smith, J., and Jones, M. (1999). How to misuse code coverage. In *ICTCS*.
- Melicher, W., Das, A., Sharif, M., Bauer, L., and Jia, L. (2018). Riding out DOMsday: Towards detecting and preventing DOM Cross-Site Scripting. In *NDSS*.
- Oberlo (2023). Browsers stats in 2023. <https://www.oberlo.com/statistics/browser-market-share>.
- Ou, H., Fang, Y., Guo, Y., Guo, W., and Huang, C. (2022). Viopolicy-Detector: An Automated Approach to Detecting GDPR Suspected Compliance Violations in Websites. In *RAID*, page 409–430.
- Pan, J. and Mao, X. (2017). Detecting dom-sourced cross-site scripting in browser extensions. In *ICSM*, pages 24–34. IEEE.
- Pilgun, A., Gadyatskaya, O., Dashevskiy, S., Zhanriarovich, Y., and Kushniarou, A. (2018). An effective android code coverage tool. In *CCS*, pages 2189–2191.
- Qian, C., Koo, H., Oh, C., Kim, T., and Lee, W. (2020). SLIMIUM: debloating the chromium browser with feature subsetting. In *CCS*, pages 461–476.
- Rapps, S. and Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE transactions on software engineering*, SE-11(4):367–375.
- Sjosten, A., Van Acker, S., Picazo-Sanchez, P., and Sabelfeld, A. (2019). Latex gloves: Protecting browser extensions from probing and revelation attacks. In *NDSS*.
- Solomos, K., Ilia, P., Nikiforakis, N., and Polakis, J. (2022). Escaping the confines of time: Continuous browser extension fingerprinting through ephemeral modifications. In *CCS*, page 2675–2688.
- SonarCloud (2023). Sonarcloud. <https://sonarcloud.io>.
- Sonarqube (2023). Automatically analyze branches and decorate pull requests. <https://www.sonarqube.org>.
- Spectral (2023). Spectral’s sast scanner. <https://spectralops.io>.
- Starov, O. and Nikiforakis, N. (2017). Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *The WebConf*, page 1481–1490.
- Sun, F., Xu, L., and Su, Z. (2011). Static detection of access control vulnerabilities in web applications. In *USENIX*, volume 64.
- Wagner, J. (2018). Trustworthy chrome extensions, by default. <https://blog.chromium.org/2018/10/trustworthy-chrome-extensions-by-default.html>.
- Wang, Y., Cai, W., Lyu, P., and Shao, W. (2018). A combined static and dynamic analysis approach to detect malicious browser extensions. *Security and Communication Networks*, 2018.
- Yerima, S. Y., Alzaylaee, M. K., and Sezer, S. (2019). Machine learning-based dynamic analysis of android apps with improved code coverage. *EURASIP Journal on Information Security*, 2019(1):1–24.
- Yu, D., Chander, A., Islam, N., and Serikov, I. (2007). Javascript instrumentation for browser security. *ACM SIGPLAN Notices*, 42(1):237–249.
- Zhao, B. and Liu, P. (2013). Behavior decomposition: Aspect-level browser extension clustering and its security implications. In *RAID*, pages 244–264. Springer.
- Zhu, H., Hall, P. A. V., and May, J. H. R. (1997). Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427.