Detecting and Understanding JavaScript Global Identifier Conflicts on the Web

Anonymous Author(s)

ABSTRACT

JavaScript is widely used for implementing client-side web applications, and it is common to include JavaScript code from many different hosts. However, in a web browser, all the scripts loaded in the same frame share a single global namespace. As a result, a script may access or even overwrite the global objects or functions in other scripts, causing unexpected behaviors. For example, a script can redefine a function in a different script as an object, so that any call to that function would cause an exception at run time.

In this paper, we systematically investigate the client-side JavaScript code integrity problem caused by JavaScript global identifier conflicts. We developed a browser-based analysis framework, JSObserver, to collect and analyze the write operations to global locations by JavaScript code. We identified three categories of conflicts using JSObserver on the Alexa top 100K websites, and detected 145,918 conflicts on 31,615 websites.

We revealed that JavaScript global identifier conflicts are prevalent and could cause behavior deviation at run time. In particular, we discovered that 1,611 redefined functions were called after being overwritten, and many scripts modified the value of cookies or redefined cookie-related functions. Our research demonstrated that JavaScript global identifier conflict is an emerging threat to both the web users and the integrity of web applications.

CCS CONCEPTS

• **Software and its engineering → Software safety**; • **Security and privacy → Browser security**.

KEYWORDS

JavaScript; Identifier conflicts; Web applications

1 INTRODUCTION

It is common to separate code of different functionalities into multiple external JavaScript files in developing web applications. Including scripts from many different hosts is also a popular practice. This allows a developer to reuse the code in other third-party programming libraries and to easily build an application rich of functionalities. For example, to include a social plugin like the Facebook Like button, a web developer needs to include only a remote script from Facebook and add two `<div>` tags in her/his web page [7].

While enhancing the functionality of an application, the included third-party scripts may cause unexpected behavior to the developer's own code. In JavaScript and many other programming languages, we use an identifier to refer to a value/object or a function in memory. In the client-side JavaScript runtime environment, *i.e.*, the web browser, there exists a single global namespace for all identifiers in scripts loaded in the same frame. Any variable or function defined in a script's own main scope is available to any other script executing in the same frame. This means that a script can not only directly call global functions and read the values of global variables in another script, but also modify its global objects or functions. Since JavaScript is a weakly typed programming language, a script can even change the type of a global variable/function without immediately causing any exceptions or errors. Such kind of global identifier conflicts can compromise the *integrity* of the developer's own code, causing it to take a different branch, return an incorrect value, or simply crash, *etc.*

Global identifier conflicts are difficult to prevent. On the one hand, to avoid identifier conflicts, a developer needs to carefully examine the source code before she/he includes a third-party script, which could be very difficult because of code minimization or obfuscation. She/he might have to change all conflicting locations in her/his own code to have a conflicting script included unless an alternative non-conflicting script can be used. On the other hand, if no conflict is found, a script might still cause a conflict in the future. The third-party code is hosted on a remote server and can be modified by the script provider at any time without notifications. The developer can enforce integrity check of a script, for example, by using Subresource Integrity [21] or the `sha256-` option of the Content-Security Policy (CSP) [34]. The application, however, can be broken whenever the remote code needs to be updated (*e.g.*, due to a security advice). Further, a script can dynamically include any other scripts, which may also contain global identifiers that conflict with the existing ones. This could be prohibited by configuring a CSP policy. However, CSP had a limited adoption rate [5, 6] because many websites require to dynamically load additional scripts from arbitrary sources, *e.g.*, if they host ad impressions that can be resold.

Prior research have studied potential global identifier conflicts between two JavaScript libraries. In [25], the authors generated synthetic clients to test if two libraries would cause different behaviors when they are loaded under different settings. Such clients can test only the simple operations of the libraries whereas the code in real applications could be much more complex. The result cannot reflect the conflicts in real applications, which may include more than two libraries. They may even include custom code that is dynamically loaded from both the developers' own hosts and the third-party hosts. Finally, the analysis is based on a selective record-replay dynamic analysis framework [33] that instruments selected source code. Thus, the tool does not cover any additional code that is dynamically loaded.

In this paper, we aim to study JavaScript global identifier conflicts in real websites. We face several challenge. First, most JavaScript code is asynchronously executed as callback. It is difficult to reason about which definition is valid when a variable is used. Second, JavaScript is a weakly typed and dynamic programming language. Static analysis is likely to overestimate the conflicts, resulting in many false positives. Third, not all code in a real website is available statically. A site can load multiple external scripts from arbitrary

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

sources dynamically at run time. Fourth, JavaScript supports object-oriented programming features. A write to a global object can be performed within the method of that object itself.

To overcome the above challenges, we develop JSObserver, a browser-based dynamic analysis framework that monitors and logs write operations to JavaScript global locations (*i.e.*, variables and functions). We perform *just-in-time code instrumentation* by modifying the Chrome V8 JavaScript engine. The instrumentation allows us to cover all code that is executed in the run time. We dynamically insert the monitoring code when a script performs an operation related to memory write. Specifically, we maintain a shadow variable to perform dynamic type inference. We log a function definition when a function is first parsed in V8 or a function literal is assigned to a variable. We log each memory write operation, including an assignment, an object definition/creation, and a function return. Since the objects in JavaScript are copied/passed by reference instead of by value, we also implement a shadow and immutable identifier property that uniquely identifies each JavaScript object in memory. This makes alias analysis much simpler and accurate. With the logs, we detect three kinds of conflicts—variable value conflict, function definition conflict, and variable type conflict. Although our analysis is not sound because we do not test all possible execution paths, it is precise as every conflict we detect must have happened.

We implemented a prototype of JSObserver based on the Chromium browser version 71, and used the prototype to analyze the main pages of the Alexa top 100K websites. We show that this class of code integrity problem is very common—overall, we found 36,813 function definition conflicts on 9,566 websites, 27,893 variable type conflicts on 3,501 websites and 81,212 variable value conflicts on 27,199 websites. The conflicts were mainly caused by the use of short/simple identifiers and duplicate inclusion of scripts. In particular, because of conflicts, cookies on 109 websites were modified and many cookie-related functions were redefined. Our research demonstrates the strong need to isolate JavaScript code from different organizations into separate namespaces.

To the best of our knowledge, we are the first to systematically measure and analyze JavaScript global identifier conflicts on a large scale. In summary, we make the following contributions.

- We develop JSObserver, a browser-based dynamic analysis framework for studying JavaScript global identifier conflicts.
- We perform an empirical study on Alexa top 100K websites and make our data publicly available.
- We characterize the detected conflicts in real web applications and discuss the security implications.

2 PROBLEM STATEMENT

In this section, we first formally define the three types of conflicts that we study, then demonstrate the scope of our research, and finally discuss our research challenges.

2.1 Definitions

We consider conflicts caused by writes by multiple scripts to the same global memory location (*i.e.*, a variable or a function) in JavaScript. A global memory location can be accessed through one or more global identifiers in any scope in JavaScript. All memory locations are properties of some object. They can be accessed through the dot notation or the bracket notation. For example, both `window.x["y"]` and `window.x.y` point to a property named as `"y"` of the object `window.x`, which is a property named as `"x"` of the global object `window`. The identifier `window` is often omitted, *e.g.*, as in `x["y"]`. Global functions are also considered as properties (or more precisely, methods) of the object `window`. We define three categories of JavaScript global identifier conflicts next.

Value Conflicts. Value conflicts happen when two or more scripts write to the same global variable with different values in the same type. For example, a script `S1` may assign `1` to the variable `state`, which is then overwritten by another script `S2` to `0`. The control flow of `S1` can be changed if it later uses this variable in a conditional statement in a callback function. Note that the writes to the same property of a global object variable, *e.g.*, `loc.x`, with different values in the same type are also considered as value conflicts.

Function Definition Conflicts. This type of conflicts occurs if two or more scripts define a global function with the same name. The runtime behavior is normally undetermined and depends on the order of the function definitions. Normally, the function defined in a script that is most recently loaded is selected by the browser.

Type Conflicts. A type conflict occurs when the same global location, *i.e.*, a variable or a function, is written by multiple scripts with values of different types. For example, `f` was defined by script `S1` as a global function. Script `S2` may assign a string to `f`, which would surprise `S1` and cause it to crash when it calls `f()` to handle some user actions. Changing the type of a property of a global object variable by a different script also constitutes to a type conflict.

2.2 Scope of Research

We focus on studying a form of JavaScript *code integrity* problem caused by global identifier conflicts. The conflicts can be caused by scripts from either a different organization, *i.e.*, the *cross-organization conflicts*, or the same organization, *i.e.*, the *intra-organization conflicts*. We study the following three kinds of cross-organization conflicts: 1) a third-party script overwrites a global variable/function defined by a first-party script; 2) a first-party script overwrites a global variable/function defined by a third-party script; and 3) a third-party organization's script conflicts with another third-party organization's script. Although a web developer may also write code that causes any of the three types of conflicts, we think this is probably the design choice of the developer. Therefore, we do not report conflicts occurred within the same script.

In our research, an organization is recognized by the domain name, excluding the top level domain (TLD), of a script's source URL. For example, scripts loaded from www.google.com and adservice.google.co.uk belong to the same organization because their domain names are both google. We acknowledge that this is not the best way to detect all domain names of a particular organization. For example, both www.twitch.tv and www.amazon.com are owned by Amazon; cdn.a.com and www.cdn-a.com may also belong to the same organization a.com. However, such cases cannot be easily detected without additional information provided by humans. We leave it as a future work to improve the method to determine the relationships of two domain names.

A conflicting script can be either directly included by the web developer, or indirectly included by another script in the same

web frame. The conflicting script has the default full privilege to access any content in its embedding frame. We do not consider code injection attacks like cross-site scripting (XSS), although XSS is another form of threats to the integrity of client-side JavaScript code. Code injection attacks are actually orthogonal to global identifier conflicts as a conflicting script already executes in the target frame.

Note that our research goal is not to determine a conflict as malicious or benign, as many of the conflicts can be caused unintentionally. Rather, we aim at detecting the conflicts that can compromise the integrity of a script and analyzing the potential security implications.

2.3 Research Challenges

We face the following challenges in detecting the conflicts.

Asynchronous Execution. JavaScript is single-threaded in the browser. Most scripts are asynchronous, *i.e.*, the execution of multiple scripts can be interleaved with callback functions. Therefore, static reaching definition analysis is imprecise because a variable defined in the main function can be asynchronously used in a callback function and asynchronously redefined by another script.

Type Inference. JavaScript is a dynamically-typed and weakly-typed programming language. An identifier can be used for data of different types without explicit type conversions. The type check is performed at run time. Therefore, a purely static analysis approach is not sufficient to detect variable type conflicts.

Object Support. Objects in JavaScript are supported by its prototype mechanism. Except for the primitive types, all other variables are of the type *object*. A script can overwrite a part (*i.e.*, a property) of an object indirectly by invoking a method of the object. We need to identify the receiver object when a write operation occurs within a method instead of a normal function. In particular, **`this`** is a commonly used identifier in JavaScript, and can point to different objects in different contexts. For example, **`this`** points to the global object `window` in a normal function scope or in the global scope; it refers to an object when it is accessed within the scope of a method of the object. In order to determine the target of a write, we need to infer precisely which object **`this`** points to.

Alias Analysis. All objects in JavaScript are copied/passed by reference instead of by value. Therefore, the same global location may be pointed by multiple identifiers in different scripts. For instance, a global variable `X` can be modified indirectly by a script that writes to the property of `Y` (*e.g.*, `Y.property = 1;`) if `Y` is an alias to `X`. Further, when passed as an argument to a parameter of a function, a global object can be modified through the parameter within the function. In other words, to detect the writes to a global location, we need to keep track of all identifiers or aliases pointing to the same object.

3 DESIGN AND METHODOLOGY

In this section, we present JSObserver, a browser-based dynamic analysis framework for detecting JavaScript global identifier conflicts at run time. We record each function definition in the V8 parser to detect function definition conflicts (§3.1). We perform just-in-time instrumentation of all JavaScript code that is executed to cover all writes to a memory location (§3.2). The records allow us to detect conflicting writes by different scripts to the same global memory locations (§3.3).

3.1 Recording Global Function Definitions

The root cause of function definition conflicts is that two or more scripts can define their functions using the same global identifier (*i.e.*, function name). Therefore, we need to find all functions that are defined in each script. A developer can define a global function in two ways: 1) defining a named function directly in the global scope, *e.g.*, **`function`** `f(args){ stmts; }`; and 2) assigning a function literal to a global variable, *e.g.*, `window.f =` **`function`** `(args){ stmts; }`. Note that a script may assign a function literal with a non-empty function name to a variable, *e.g.*, **`var`** `x =` **`function`** `f(args){ stmts; }`. That function name (*e.g.*, `f`) is an invalid identifier (*i.e.*, `undefined`) in JavaScript.

Finding the first type of function definition is not difficult. In the V8 engine, the parser needs to first parse the JavaScript code in the global scope before the compiler outputs the target code. Therefore, JSObserver logs all global functions with a non-empty name. The log includes a unique ID (*e.g.*, timestamp), the position of the function definition, an ID of the script, and an ID of the execution context. The timestamp-like log ID enables us to understand at each point of time which definition is valid. We can also leverage it to study variable type conflicts involving a global function. In particular, we can determine if a function is changed to an object or if an object is changed to a function.

Finding the second type of function definition requires us to also monitor writes to global variables. We discuss it next.

3.2 Recording Writes to Variables

In this section, we describe how JSObserver logs the write operations to memory locations, *i.e.*, variables. A JavaScript variable can be written in primarily two ways: 1) directly written by assigning a value to it or by coping another variable to it; or 2) partially written by assigning a value or by copying another variable to one of its properties if the variable being written is an object. The value that can be written to a variable or its properties can be in several categories: 1) a primitive type value (*e.g.*, a number or a boolean value); 2) an object literal (*e.g.*, `{a:1, b:0}`); and 3) an object initialized with a constructor (*e.g.*, **`new`** `Person("John", 20)`). However, capturing all the writes to a variable is very challenging.

First, variables in different scopes can use the same identifier (name). We need to differentiate local variables from global variables to avoid over estimation. To tell if an identifier `v` is a global variable or not, JSObserver checks the scope of the current statement `S` where `v` is used. If the current scope is the *global scope*, the identifier in `v` is global and is directly logged. If the current scope is a *function scope*, JSObserver would search the identifier `v` in the parameter list and declaration list of the current function. If `v` is found in the lists, it is determined as a local name; otherwise JSObserver continues to search the lists of an outer function scope until either a match is found or it reaches the global scope. A special situation is that the current scope is an object scope or a function scope in an object (*i.e.*, a method) and the identifier points to a property of the current object, *e.g.*, **`this`**`.p`. This requires us to infer which object the keyword **`this`** points to, which we describe next.

Second, variables of non-primitive types, *i.e.*, objects, are copied or passed by reference instead of by value in JavaScript. In order to detect the writes to the same object, we need to identify all the *valid* aliases to it. One intuitive approach is to keep track of all the assignments involving an object or a variable of an object. However, this approach is error-prone because an object can be passed into several nested function calls or assigned as a property of another object. An object variable `v` can also be assigned to another object `obj` (*e.g.*, `v = obj`), hence becomes a new alias to `obj`. Further, an object can be self-referenced in its methods with the keyword **`this`**, which can potentially point to any object. To solve this challenge, JSObserver maintains a unique and immutable shadow ID property– `__id__` –of each JavaScript object in V8. Whenever an object is being written, we can identify it with this shadow ID property, regardless of the JavaScript variable name being used.

To record the type of a variable, we leverage the **`typeof`** operator in JavaScript. However, if its operand is an expression instead of a simple identifier, the expression would be evaluated again when we infer the type using it. This would cause some unexpected behavior. For example, consider the assignment statement `arr[i++] = f()`. To log the type of the memory write destination, we can use **`typeof`** `arr[i++]`. This would cause an additional update of the value of `i`, such that the type of the wrong memory location is returned. To avoid this kind of side effects, we introduce a shadow variable `v'` for each direct write operation to a variable in our instrumentation. The write is applied to the original variable first, and then applied to the shadow target in a nested assignment statement. JSObserver records the type of the write target by specifying the shadow variable as the operand of **`typeof`**.

We next discuss in detail the instrumentations that JSObserver performs for each type of write operations to a memory location: 1) assignment statements; and 2) object literal and constructor expressions. We summarize the instrumentations in Table 1. When any of the write operations is executed, JSObserver infers and records the type of the memory write target `v`, the value of the target if it is a primitive type variable, the expressions `e` in the operation, a unique log ID, and the IDs of the script and the execution context, using a custom function **`recordWrite`**`(v, s, t, e)`. Inside the function, it infers the type of the write target `v` by evaluating **`typeof`** `t`, where `t` is a (shadow) variable whose type is identical to that of `v`. It logs the shadow ID property `s.__id__` where `s` is an alias to the write target object. If the target is of a primitive type, the variable `t` is passed to `s` and the function instead logs the value of `t`. The write source expression `e` is also recorded.

3.2.1 Assignment Statements. Assignment statements are the most direct way that a script can write to a variable. For each direct write target `v` in an assignment statement where `v` is a variable, JSObserver creates a shadow variable `v'` for it automatically. In particular, JSObserver replaces the assignment statement with a nested assignment statement which writes to both `v` and `v'`, as the first rule shown in Table 1. This avoids evaluating an expression like `arr[i++]` twice.

For a shorthand operator, *e.g.*, +=, JSObserver also creates a shadow variable `v'` for the direct write target `v`, as the second rule in Table 1. In case that a nested assignment statement is found, JSObserver would visit the abstract syntax tree (AST) of the nested assignment statement and replace each assignment statement node individually (rule #3 in Table 1).

Table 1: Instrumentation for recording write operations.

1.	$v = e$	$\Longrightarrow$	$v' = v = e$ $recordWrite(v, v', v', e)$
2.	$v += e$	$\Longrightarrow$	$v' = v += e$ $recordWrite(v, v', v', e)$
3.	$v_1 = v_2 = e$	$\Longrightarrow$	$v'_1 = v_1 = v'_2 = v_2 = e$ $recordWrite(v_2, v'_2, v'_2, e)$ $recordWrite(v_1, v'_1, v'_1,$ $\quad v'_2 = v_2 = e)$
4.	$v.p = e$	$\Longrightarrow$	$v' = v$ $v'.p = e$ $recordWrite(v.p, v', v'.p, e)$
5.	$\{p_1 : e_1,$ $\quad p_2 : e_2, \ldots\}$	$\Longrightarrow$	$o' = \{p_1 : e'_1 = e_1, p_2 : e'_2 = e_2, \ldots\}$ $recordWrite(o'.p_1, o', e'_1, e_1)$ $recordWrite(o'.p_2, o', e'_2, e_2)$
6.	$new\ Obj(\ldots)\{$ $this.p = e;\}$	$\Longrightarrow$	$new\ Obj(\ldots)\{$ $o' = this;$ $o'.p = e;$ $recordWrite(this.p, o', o'.p, e);\}$

We need to also identify an object when it is partially written through its property. We could again try to leverage the above shadow variable to get the shadow ID to avoid re-evaluation of an expression, as in `v' = v.p = e`. However, the shadow variable would be an alias to the source object `e` being assigned to the property instead of an alias to the target object `v`. Further, if the write source's type is a primitive type, the shadow variable `v'` would be a value copy of it instead of an alias to it.

In such a situation where the write target is a property of an object variable, *e.g.*, `v.p = 1`, JSObserver creates a shadow variable `v'` of the parent object variable `v` instead of its property `v.p`. It then applies the write to the property through the shadow variable instead of the original variable, *e.g.*, `v'.p = 1`, to avoid expression re-evaluation (rule #4 in Table 1). As a result, we are able to identify property writes to the same object. For example, a method writes to one property of the owner object through **`this`**`.p = 1;`. JSObserver will transfer the code into `v' = this; v'.p = 1;`, and identify the owner object through the value `v'.__id__`. One special case is that the property is also an object. Such a write would not modify that object represented by the property, which was essentially an alias, but make the property either a new alias to another object being assigned to it or a primitive type variable. Therefore, we do not create a separate shadow variable for an object property. We do check the type of the property (*e.g.*,**`typeof`** `v'.p`) rather than that of the shadow object to detect type conflicts.

3.2.2 Object Literal and Constructor Expressions. To detect a write conflict to a property of a global object, JSObserver needs to record all writes to it, including the initial definition. A property can be defined in two ways. First, the property is directly initialized in an object literal expression, as rule #5 in Table 1. JSObserver creates a shadow variable `o'` for the newly created object, and calls **`recordWrite`** to record the write to each property of the object. The

unique object ID would also be logged with the shadow variable `o'`. Second, a property `p` may be defined within the constructor or a method of an object through the identifier **`this`**, as rule #6 in Table 1. In the case of an object constructor, JSObserver also creates a shadow variable `o'` for the object inside the constructor. JSObserver then logs the write to the property **`this`**`.p` and also the shadow ID of **`this`** through `o'`.

With the help of the unique object shadow ID, we avoid the burden of tracking the aliases to an object. In our analysis stage, we are able to search backward in the logs to find all write records with the same object shadow ID to detect any write conflicts.

3.3 Detecting Conflicts

In this section, we discuss how we leverage the records collected by JSObserver to detect the three types of global identifier conflicts.

3.3.1 Function Definition Conflicts. The detection of global function definition conflicts is very straightforward. We need to simply check the function definitions in each frame to find if the same global function had been defined for more than once by different scripts. However, a global function can also be defined by assigning a function literal to a global identifier. To detect conflicting function definitions by function literal writes, we also find assignment logs where the type of the write target is *function*, and search the target identifier, *i.e.*, the function name, in the function definition logs.

3.3.2 Value Conflicts and Type Conflicts. If a global variable is of a primitive type, it does not have an alias. We will search any other write records to the same global identifier. If the logged values in two records are different and the writes are performed by two different scripts, we report it as a variable value conflict. However, if in one record the type of the global variable is different, we report it as a variable type conflict.

If a global variable is an object, a value conflict may happen in two situations. First, the variable `v` itself is overwritten with another variable. This can be easily detected by searching only the assignment records to the same identifier `v`. We do not need to check if one of its aliases is overwritten because that would effectively invalidate this variable `v` as an alias. Second, a property of the object is written. This can be detected by searching the write records of all the object's valid aliases with regards to the current assignment. If the types of the property in two writes are the same, we report it as a variable value conflict if either the type is object, or the type is a primitive one but the values are different. Otherwise if the types differ, we report it as a variable type conflict. Note that if the conflict is caused by the same script, we do not report it.

Variable and Function Type Conflicts. We find that a special type of conflicts may occur, *i.e.*, a global identifier is used as both a global function name and a global variable name. We call it as variable and function type conflicts. For example, `f` could be defined by a script as a global function. Another script may assign a primitive type value or an object to `f` either before or after this function definition. Similarly, a global variable `v` of either a primitive type or an object, may be assigned with a function literal by another script, as in `v = function (){...};`. In order to detect this kind of type conflicts, we need to also cross check the function definition logs and variable write logs. In particular, for each identifier that is defined as a global variable, we search it in the function definition logs as well as the variable write logs to determine if it is also ever defined as a function.

3.4 Implementation

We implemented a prototype of JSObserver based on Chromium version 71 using about 4K lines of C++ code. We modified the V8 parser to record global function definitions. We modified the V8 bytecode generator to add our instrumentation code for recording writes to memory locations. The `asgLogs` is implemented in the WebKit layer as a property of the `DOMWindow` class. The logs are dumped into the file system on the fly. We plan to release the source code of our prototype implementation publicly.

4 EVALUATION

In this section, we first describe the data collected in our web crawling (§4.1), then characterize the detected global identifier conflicts by demonstrating what type of conflicts are generated (§4.2) by which scripts (§4.3). Further, we provide case studies (§4.4) and analyze the affected websites and possible reasons of conflicts (§4.5). Finally, we measure the performance of JSObserver (§4.6).

4.1 Dataset and Availability

We crawled data from the main pages of Alexa top 100K websites using JSObserver in October 2019. For each website, we recorded the writes to all identifiers and the calls to functions within 120s in an assignment log file, and stored the definitions of global functions in a function definition log file. Excluding those that timed out or crashed in our data collection process, we successfully gathered assignment log files from 79,083 (79.08%) websites and function definition log files from 80,566 (80.57%) websites.

The collected data in total require about 3 TB of disk space. As a result, we currently are not able to release all the data on the Internet. Instead, we randomly sample 100 websites for each of the three categories of conflicts, and another 100 websites on which we did not detect any conflict. The log files collected for the 400 websites are available at https://drive.google.com/open?id=1yaPhJzjermysDBO2k-XV25ekuGL39KSv (about 633 MB). We will be glad to provide additional data upon requests.

We plan to make our data archived open data upon acceptance.

4.2 Category of Conflicts

In this section, we characterize JavaScript global identifier conflicts based on the categories that we define in §2.1.

4.2.1 Function Definition Conflicts. In total, we detected 36,813 function definition conflicts on 9,566 websites. In particular, there are 1,065 such cases where a function literal was directly assigned to an identifier, which was used as a function name by another script. We consider this as a special type of function definition conflicts.

We then characterize these conflicts based on the class of the conflicting scripts, *i.e.*, third party or first party. We currently are not able to determine the class of a dynamically created inline script because it does not have a source URL. The class is "unknown" for conflicts involving such inline scripts. Excluding them, we were able

Table 2: Categorization of global identifier conflicts

Category	#Websites	#Conflicts	%Conflicts
Function Definition	9,566	36,813	25.23
third -> first	715	1,510	1.03
third -> diff_third	311	543	0.37
first -> third	349	704	0.48
third -> same_third	1,283	7,086	4.86
first -> first	6,829	25,580	17.53
unknown	891	1,390	0.95
Variable Type	3,501	27,893	19.12
third -> first	338	556	0.38
third -> diff_third	156	206	0.14
first -> third	288	434	0.30
third -> same_third	434	820	0.56
first -> first	1,881	22,882	15.68
unknown	643	2,995	2.05
Variable Value	27,199	81,212	55.66
third -> first	7,128	8,582	5.88
third -> diff_third	5,302	7,476	5.12
first -> third	2,021	2,493	1.71
third -> same_third	4,270	9,302	6.37
first -> first	11,986	40,248	27.58
unknown	7,980	13,111	8.99

to categorize 35,423 (96.22%) conflicts. Table 2 lists the breakdown of these function definition conflicts.

Cross-organization and Intra-organization Conflicts. Overall, we detected 2,757 (7.49%) cross-organization function definition conflicts. In particular, 543 (1.48%) cases were caused by a third-party organization's script that redefined functions of other third-party scripts. We also discovered 1,510 (4.10%) conflicts where a third-party script modified a function defined by a first-party script, and 704 (1.91%) cases where first-party scripts overwrote functions defined by a third-party script. This indicates that scripts could break the integrity of other scripts loaded from a different organization by overwriting the function definitions.

The majority (32,666 or 88.73%) of function redefinitions were intra-organization conflicts. 25,580 (69.49%) of them were caused by first-party scripts overwriting other first-party scripts, and the conflicting definitions were usually similar. We think this is not a good coding practice.

Conflicts on Browser Internal Objects. Interestingly, we detected 24 function definition conflicts on browser internal objects, and 3 of them are cross-organization conflicts. For example, a third-party and a first-party script assigned different function literals to `self.onerror`. `self` is a reserved property of the `window` object and points to the current window. The third-party script therefore redefined the way that runtime errors are handled. These conflicts obviously break the integrity of scripts from other organizations. We also found 21 intra-organization function definition conflicts on the internal objects. In particular, 16 of them were found on methods of internal objects. For instance, `document.write()` and `document.writeln()` were modified on 6 and 4 websites, respectively. We believe these are the development choices, as the scripts were all loaded from the first-party domain. However, exposing the builtin methods to all scripts is dangerous, because the scripts could change the default behavior of the embedding websites.

Duplicate Function Definition and Duplicate Script Inclusion. We identified several conflict cases where the two function definitions were identical, which we call *duplicate function definition*. This might happen when the same script is included for multiple times. In total, we found 10,151 (27.57%) cases that the conflicting function definitions were actually identical, and 9,198 (24.99%) such conflicts were detected between scripts of the same organization. The other 953 (2.59%) duplicate definitions were probably caused by including the same libraries hosted by different organizations.

Table 3: Top duplicate functions.

Function	#Websites	#Conflicts
gtag	450	631
_typeof	37	273
ez_getQueryString	235	241
getCookie	86	184
_classCallCheck	33	156

Table 4: Top redefined functions.

Function	#Websites	#Conflicts
getCookie	128	155
AKSB.done	115	115
AKSB.measure	115	115
AKSB.mark	114	114
_typeof	34	56

Table 3 lists the top functions of which we detected a duplicate definition. As can be seen, some commonly used functions (*e.g.*, `gtag()` and `getCookie()`) were repeatedly defined with the same code. One possible explanation is developers usually would not check if a script has been loaded before loading it twice. In total, we found 6,230 (16.92%) function definition conflicts caused by duplicate inclusion of scripts. Duplicate inclusion does not necessarily break the functionality of the embedding pages, but still could cause unexpected behaviors, *e.g.*, invoking the same function for multiple times. Therefore, we still consider this as a bad coding practice.

We do realize that our method to identify duplicate function definitions is not comprehensive, since a function could be implemented in various ways. For example, `a && b = 1` is functionally equivalent to `if(a){b = 1}`. However, it is not our main focus to thoroughly compare the behavior of different functions, which cannot be easily decided.

Top Redefined Functions. After excluding identical function definitions, we obtain the top five functions that were most frequently redefined by scripts from different organizations, as listed in Table 4. Note that the top function is related to cookies. By overwriting cookie-related functions, a malicious third-party script could expose a website to security risks, which we will demonstrate later. Meanwhile, the top functions that are properties of global object `AKSB` were all redefined by one third-party script https://ds-aksb-a.akamaihd.net/aksb.min.js. Further investigation shows that the first-party definition of these functions contained very simple code, *e.g.*, writes to an array. The third-party script modified them to record several important events *e.g.*, `domContentLoadedEventEnd`. For these conflicts, we think the first-party developers included some initial code snippet provided by the Akamai SDKs (*e.g.*, AKSB), which would update their code at runtime.

Call-after-redefinition. To estimate the impact of the function definition conflicts, we further searched in our logs to check if the functions were called after they were redefined by a different script. Table 5 gives the breakdown of the calls to the redefined functions. Specifically, we detected 39 cases where third-party defined functions were overwritten by first-party scripts, and were called by the

Table 5: Categorization of calls to redefined functions.

Category	#Calls	#Conflicts	%Calls
third -> first	72	1,510	4.77
third -> diff_third	16	543	2.95
first -> third	66	704	9.38
third -> same_third	252	7,086	3.56
first -> first	1,091	25,580	4.27
unknown	114	1,390	8.20

third-party scripts later. We also found 29 cases where the first-party scripts called a function that had been overwritten by a third-party script. In particular, 12 of them are cookie-related functions. One example was detected on the website http://footdistrict.com/, where a function `createCookie()` was firstly defined in a first-party script, but then overwritten by a third-party script. After that, the function `createCookie()` was called by the first party. Such a conflict could introduce severe security risks, as a malicious third-party script could manipulate the value of cookies to force a user to use the attacker's session.

Summary. We discover that function definition conflicts exist on over 9K popular websites. 2,757 (7.49%) conflicts were caused by scripts from different organizations, and 1,611 functions got called after being redefined by a different script. We found 16 conflicts on browser builtin methods, and some cookie-related functions were frequently redefined, which could be exploited to expose the websites and their users to security risks.

4.2.2 Variable Type Conflicts. We detected 27,893 variable type conflicts across 3,501 websites. The results are presented in Table 2.

Cross-organization and Intra-organization Conflicts. We found 1,196 (4.29%) variable type conflicts caused by scripts from different organizations. We also detected 434 (1.56 %) cases where a first-party script modified the type of a third-party defined variable. This shows that scripts in any class could be affected by variable type conflicts, which can introduce potential risks as inconsistent types could cause exceptions and lead to behavior deviation.

Meanwhile, most (23,702 or 84.97%) of variable type conflicts were intra-organization conflicts. We believe these conflicts could be the design choice of the developers because the conflicting scripts were loaded from the same organization's domains. However, they could also cause difficult-to-detect bugs if the developers of conflicting scripts were not aware of the conflicts.

Variable and Function Type Conflicts. We detected 971 (3.48%) special type conflicts where a primitive value or an object was assigned to an identifier before or after the identifier was used as a global function name. Specifically, 3 of them were directly declared as a function. For example, the variable `isChrome` was defined as `true` in a first-party script on https://www5.javmost.com/ before a function was assigned to it. We detected 157 (0.56%) type conflicts that a function was modified to be a non-function variable. As a result, any call to them would cause a runtime exception. We also found one special case on the website http://vbspu.ac.in/, where a first-party script modified `dropdown` to an object, after declaring it as a function itself. Such a conflict caused a `TypeError` when `dropdown` was called later. This could be an implementation error.

Further inspection showed that 106 out of 157 type conflicts involved identifiers with a length less than 3, *e.g.*, `ma`, `NN`, *etc.* This suggests that developers should use unique longer and meaningful names to avoid the type conflicts.

Table 6: Cookie value conflicts.

Category	Same Organization		Diff Organization	
	#Conflict	#Website	#Conflict	#Website
Value Modification	97	90	19	19
Addition	2,987	2,217	7,625	5,428
Deletion	5	4	0	0

Summary. We detected variable type conflicts on over 3K websites, and 1,196 (4.29%) were caused by scripts of different organizations. Especially, global functions could be redefined as non-function variables, which could cause run-time errors when they are called. These conflicts were mainly caused by scripts using simple function names, *e.g.*, `NN`. It indicates that the developers should choose unique names to avoid the conflicts.

4.2.3 Variable Value Conflicts. We detected 81,212 variable value conflicts across 27,199 websites. Interestingly, we detected 10,733 (13.22%) cookie-related variable value conflicts, as shown in Table 6.

Cross-organization and Intra-organization Conflicts. We detected 18,551 (22.84%) cross-organization variable value conflicts, and 8,582 (10.57%) of them were caused by third-party scripts overwriting first-party variables. For example, `ga.l` was modified from `...437` to `...447` by a third-party script on the website https://filmow.com/. Meanwhile, we found 2,493 (3.07%) cases where a third-party defined variable was modified by first-party scripts. We also found 49,550 (61.01%) value conflicts that were caused by scripts of the same organization, and most of them (40,248 or 49.56%) were caused by first-party overwriting first-party variables. We think these are the development choices.

Conflicts on Browser Internal Objects. In total, we found 228 cases that multiple scripts wrote to cookies with the same name. In particular, 116 conflicts caused existing cookie values to be modified, as the conflicting scripts assigned different values to cookies with the same name, path and domain, and 19 of them were caused by scripts from different organization. One example is on the website https://www.betfair.it/, where a third-party script from ie1-sscbf.cdnppb.net overwrote the value of cookie `bfsd` from `...39065` to `...41129`. In this example, the cookie value was originally defined by a first-party script. We believe this is a privilege abuse. We also found 97 cases that cookie values were modified by scripts of the same organization. Although it shall be legitimate for a first-party script to modify a cookie, we think third-party scripts shall not modify cookies which all belong to the first party site.

Moreover, we detected 5 cases that scripts removed an existing cookie by setting the expire time to a past one. One example was found on website https://www.proporta.com/, where a third-party script removed an existing cookie `"_gcl_au"` that was originally defined by another third-party script from googletagmanager.com. We believe the above example is legitimate as Civic Computing provides a service for cookie compliance under GDPR [4]. However, removing an existing cookie may usually cause the server-side program to malfunction.

In addition to `document.cookie`, we also found 52 variable value conflicts on other internal object properties, and 7 of them are cross-organization conflicts. One example was found on website http://

popcornnowis.blogspot.com/, where a third-party script overwrote `window.name` defined by another third-party script loaded from a different domain.

Boolean Value Conflicts. We also detected 1,258 boolean value conflicts on global identifiers, which were used to control the program behavior. For instance, on website https://olxliban.com, a first-party script defined a variable `_adblock` as `true`, then a third-party script https://olxbstatic-a.akamaihd.net/.../advertising.js?... modified it to `false`. This variable was used to represent the existence of an ad blocker. In the case it is `false`, an advertising script from doubleclick.net would be included. However, this can be abused by other third-party scripts. For example, a script can always set `_adblock` to true to prevent the injection of ads from Google DoubleClick and to inject its own ads instead.

It reveals that because of variable value conflicts, the control flow of JavaScript code and the appearance of the embedding page could also be influenced. Although HTML elements can be directly modified using JavaScript, it is much more difficult to reason about the indirect modification caused by global identifier conflicts.

Summary. We detected 81,212 variable value conflicts on over 27K websites. 116 conflicts caused an existing cookie to be modified. Also, 27 conflicts were caused by third-party scripts overwriting internal object properties. We believe they are privilege abuse. Some first-party defined boolean values were modified by third-party scripts, resulting in different execution paths of the first-party program. The findings demonstrate that variable value conflicts could change program behaviors.

4.3 Conflicting Scripts

We characterize the global identifier conflicts based on the conflicting scripts, with a focus on the cross-organization conflicts.

4.3.1 Function Definition Conflicts. We detected 961 unique scripts redefining a function of another script from a different organization. As shown in Table 7, the top script doczy_full-1570133213.min.js redefined 125 third-party functions on the website https://modeloinicial.com.br/, which included a duplicate script from a third-party CDN host cdn.plune.com. The situation for other top scripts was similar. Even though the conflicting function definitions were identical, including the same code for multiple times could result in undesired effects and should be avoided.

The most prevalent scripts causing function definition conflicts are listed in Table 8. Similar to the top script aksb.min.js that we had discussed in §4.2, the script containr.js redefined `window.mpfContainr()`, which was originally a very simple function defined by the first party. In contrast, the scripts from exoclick.com and wololo.net contained duplicate definition of functions, which have been defined by first-party scripts. Further investigation shows that on some websites (*e.g.*, https://www.stileproject.com), the script from exoclick.com was indirectly included by other third-party scripts, and it contained identical code as the script it overwrote. We think that these conflicting scripts were operated under different domain names but might actually belong to the same organization, and the conflicts were caused by unexpected duplicate inclusion.

In contrast, the script from static.tacdn.com indeed changed the definition of function `window.uiOverlay()` on 23 websites that all came from tripadvisor.com. We think static.tacdn.com might be a CDN host of TripAdvisor. However, as shown in Listing 1 and Listing 2, the redefined function does not wait until the DOM has been fully loaded before proceeding. Such a conflict could cause different behaviors on the embedding page and should be avoided.

```
window.uiOverlay = function(l) {
  if (document.readyState in {complete: 1, loaded: 1}) {
    require(["trjs!overlays/uiOverlay"], function(e) { e.apply(
      null, i); })
  }
  else {
    document.addEventListener("DOMContentLoaded", function() {
      ... uiOverlay.apply(null, e); })
  }
}
```

Listing 1: Original definition of `window.uiOverlay()`.

```
window.uiOverlay = function() {
  require(["overlays/uiOverlay"], function(e) { e.apply(null, t
      ) });
}
```

Listing 2: Redefinition of `window.uiOverlay()`.

4.3.2 Variable Type Conflicts. We detected 577 scripts causing cross-organization type conflicts. The top five scripts are also listed in Table 7. Four of them are first-party inline scripts that redefined variables `aj_adspot`, `aj_zone` and `aj_server`, *etc.*, from `undefined` to a certain value. These variables were originally defined by the third-party scripts, and were eventually used to determine the advertisements to inject. This indicates that by overwriting certain custom variables, a "malicious" script could indirectly control important DOM elements, *e.g.*, ads, that are shown to the visitors.

The noticeable script https://blogroll/livedoor.net/js/blogroll.js in Table 8 modified variable `blogroll_channel_id` from a number to `undefined` on 63 websites, after using it as the ID of an injected `<div>` element. This reveals that variables could be modified by other scripts to `undefined`, causing exceptions when referenced.

4.3.3 Variable Value Conflicts. In total, we detected 7,419 unique scripts modifying the value of variables defined in scripts of other organizations. The most commonly included script came from facebook.net. It was detected to modify the value of `fbq.version` from `"2.0"` to `"2.9.4"` on 2,541 websites. `fbq.version` was initially defined by the first party scripts, and was used to determine the version of a facebook script included on the embedding websites. In this case, we believe the developers intentionally allowed fbevents.js to update the version number.

In addition, three of the top scripts were third-party scripts that added new cookie values. Similarly, three of the top prevalent scripts modified the value of cookies on hundreds of websites. For instance, the script from cr.acecounter.com added a new cookie `"ACEFCID"`, whose value was used by the first-party website to determine the source URL of an image on that page. This suggests that cookies are common write targets of different scripts and it is risky to allow third-party scripts to set the cookies of a first-party website.

4.4 Case Studies

In this section, we discuss several interesting conflicts detected by JSObserver to further demonstrate the potential risks.

Redefining Cryptography Functions. We also found cases where critical cryptography functions were defined by multiple scripts. For

Table 7: Top scripts overwriting global identifiers.

Conflict Category	Script URL	#Conflict Targets
Function Definition	https://modeloinicial.com.br/js/doczy_full-...min.js	125
	https://www.tcm.com/js/ads/cnn_adspaces.js	44
	https://analytics-eu.clickdimensions.com/forms.js?_=...	41
	https://tmui2k8.fs.ml.bac-assets.com/sve/js/v4/ms/Microsoft_4.0_min...js?	31
	http://cdn-5.wololo.net/wagic/wp-content/cache/minify/ae569.default...	20
Variable Type	http://www.acolyer.org/	13
	https://www.engineersgarage.com/	10
	https://www.massdevice.com/	10
	https://www.asce.org/	10
	https://harpers.org/	10
Variable Value	http://cr.acecounter.com/Web/AceCounter_AW.js?gc=BH3A414...	44
	http://cr.acecounter.com/Web/AceCounter_AW.js?gc=AS4A405...	41
	http://sas.nsm-corp.com/sa-w.js?gc=...	40
	https://www.informationweek.com/Default.asp	16
	http://ads.bumq.com/ad_show2.js	12

Table 8: Top prevalent conflicting scripts.

Conflict Category	Script URL	#Website
Function Definition	https://ds-aksb-a.akamaihd.net/aksb.min.js	111
	https://cdn.mookie1.com/containr.js	47
	https://ads.exoclick.com/ads.js	32
	https://static.tacdn.com/js3/build/concat/short_lived_global-c-...js	23
	https://cdn-gae-ssl-default.akamaized.net/js/isp.v.2.0.1.min.js?v=...	23
Variable Type	https://blogroll.livedoor.net/js/blogroll.js	62
	https://ads.exoclick.com/ads.js	38
	https://s-pt.ppstatic.pl/o/js/osnowa.js?...	22
	https://ads.exosrv.com/ads.js	16
	http://1.citynews.stgy.ovh/ shared/scripts/3rdp-censor/fab.js	14
Variable Value	https://connect.facebook.net/en_US/fbevents.js	2,541
	https://sb.scorecardresearch.com/beacon.js	1,473
	https://top-fwz1.mail.ru/js/code.js	596
	https://ssl.google-analytics.com/ga.js	260
	https://secure.quantserve.com/quant.js	243

example, the website https://www.clublexus.com/ contained duplicate definitions of several cryptography functions, *e.g.*, `b64_hmac_md5()`, `binl2b64()` and `core_hmac_md5()`, *etc.* Our inspection shows that this website included two scripts that contain identical definitions of these functions. However, a third-party script can modify `core_hmac_md5()` to generate fake message authentication code (MAC) to break data integrity.

Conflicting Adsense Publisher IDs. Interestingly, we detected on several websites that first-party inline scripts wrote different Adsense publisher IDs. For example, on the website https://www.acillust.com/, the developer's scripts wrote two different values `ca-pub-5938...` and `ca-pub-6219...` to the same global identifier `google_ad_client`. It is used as a unique identifier of a publisher's Adsense account to distribute the advertising revenues. In this case, we believe that first-party developers may have two different Adsense accounts. However, a malicious script could steal the advertising revenues by simply replacing the value of `google_ad_client` with her/his own publisher ID.

4.5 Affected Websites and Reason of Conflicts

4.5.1 Affected Websites. We computed the rank distribution of affected websites and found that the conflicts caused by scripts from both first-party and third-party were uniformly distributed.

In particular, we detected conflicts on some top ranked websites. One of the top affected websites is https://www.amazon.com/. It included two scripts that assigned different callback functions to the onevent handler `window.onerror` using `window.onerror = function(){...}`, which would directly replace any existing event handler [20]. The second script therefore redefined the way how runtime JavaScript errors were handled on Amazon. Our inspection shows that the two scripts were both first-party inline scripts,

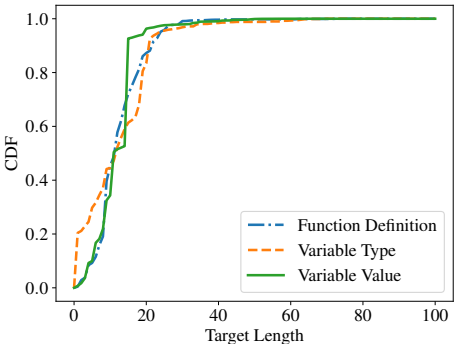

Figure 1: Conflict distribution regarding conflict targets length.

and the second definition actually calls the original event handler. We believe that this is an implementation error. The developer could have used `window.addEventListener("error", function(){...})` to avoid replacing all existing event handlers. Another example is http://babytree.com/, whose Alexa ranking was 79. The value of a global variable `a` defined by a first-party script was modified from an object to a string by another first-party script from the same domain. We consider this as a bad coding practice, which might result from the use of simple variable names.

Our study indicates that all websites, including the top ranked ones, are subject to JavaScript global identifier conflicts.

4.5.2 Possible Cause of Conflicts. We derived the distribution of cross-organization conflicts regarding the length of conflict targets. As shown in Figure 1, for all categories of conflicts, shorter conflict targets were much more likely to be overwritten by other scripts. In particular, 318 (11.53%) function definitions conflicts, 378 (31.61%) variable type conflicts and 3,103 (16.73%) variable type conflicts were detected on variables/functions whose names contain no more than 6 characters. The top redefined identifiers include `_tmr`, `_hsq`, and `a`, *etc.* This suggests the use of short and common names is one possible cause of global identifier conflicts, and the developers should assign longer and unique names to avoid the conflicts.

As discussed in §4.2 and §4.3, duplicate script inclusion is another cause of function definition conflicts, and is actually not rare (it caused 16.92% of function definition conflicts). Web developers should be careful when including scripts, as duplicate inclusions could cause undesired effects.

4.6 Performance of JSObserver

We measure the slowdown on page loading time to evaluate the performance overhead incurred by JSObserver. Specifically, we used a Vanilla Chromium browser and the prototype of JSObserver to visit the Alexa top 100 websites separately, waited for at most 5 minutes before closing the browser, and calculated the average page loading time and the average slowdown in three rounds. The experiment results are shown in Table 9. As shown, JSObserver incurs an average slowdown of 10.96X, and the maximum slowdown is around 200X on https://www.youtube.com, which included 35 scripts. This shows that JSObserver incurs lower overhead compared with similar tools, *e.g.*, Jalangi [33]. However, we also observed that several pages did not finish loading within 5 minutes. The reason is we

Table 9: Slowdown on Page Loading Time.

Round	Average (X)	Max (X)	#Incomplete Loading
1	10.84	192.48	2
2	11.58	194.78	6
3	10.45	213.62	4

injected multiple instructions for each simple write to a memory location. We will further discuss in §5.

5 DISCUSSION

We discuss the limitations of our current work, the possible mitigation of the identifier conflict threat, and our future work.

Overhead and Coverage. JSObserver introduces high performance overhead as it injects several instructions for each write to a memory location. Therefore, many websites could not finish loading within 2 minutes in our experiment. Our goal was not to detect all possible conflicts in real time. Instead, we tried to detect as many as we could. Indeed, we were still able to gather many logs generated within the timeout and reveal the global identifier conflict problem in the real world. We plan to optimize the performance of JSObserver to improve the efficiency of data collection.

Script Isolation. A script can access and overwrite the global identifiers in another script due to the fact that all the scripts included in the same frame share the same global namespace. This indicates that the global identifier conflict problem can be solved by isolating the scripts that write to the same memory location. Therefore, we can leverage existing browser isolation mechanism to isolate different JavaScript code in separate execution environments. However, this may introduce significantly high overhead at run time and break the functionality of code that depends on each other. We leave it as our future work.

Avoiding Common Identifiers. Our analysis shows that many scripts used simple or popular identifier names, *e.g.*, `i`, `getCookie`, *etc.*, which is another cause of conflicts. Therefore, the problem might also be mitigated by ensuring that the same identifier is not used by multiple scripts. For example, we can statically or dynamically instrument JavaScript code by appending different random strings to the identifiers in different scripts. We plan to implement and evaluate such a mechanism in the future.

6 RELATED WORK

JavaScript Conflict Analysis. Patra *et al.* [25] proposed ConflictJS, an automated approach to analyzing the conflicts between JavaScript libraries. However, they only studied a limited number of JavaScript libraries in a synthetic environment. In real world applications, there could be more than two JavaScript libraries. Further, ConflictJS is built on top of Jalangi, which is a dynamic JavaScript analysis framework based on selective record-replay technique [33]. Therefore, they are not able to detect the conflicts in dynamically loaded code. In contrast, JSObserver is able to detect the conflicts between any scripts, including those that are dynamically loaded.

JavaScript Type Inference. Pradel *et al.* [26] proposed TypeDevil to detect identifiers that have inconsistent types. Jensen *et al.* [15] proposed a static analysis framework for JavaScript and implemented an analysis prototype on top of [24]. Hackett *et al.* [11] presented a hybrid type inference approach for JavaScript based on points-to analysis. These works focus on inferring JavaScript type information within a single script. However, our dynamic analysis framework aims to detect the type inconsistency of global identifiers across different scripts. Meanwhile, there have been several learning-based approaches to predicting the type for JavaScript code [12, 19, 28]. They aimed to statically infer about a variable type and therefore enable the generation of much faster code, which is orthogonal to our work. In our work, we leverage the JavaScript built-in type checker to infer the type of a variable at run time.

Cross-domain Script Inclusion. Yue and Wang [35] studied several insecure practices regarding JavaScript, including duplicate inclusion. Ratanaworabhan *et al.* [27] and Richards *et al.* [31] analyzed the behavior of popular JavaScript libraries. Richards *et al.* [30] focused on the security risks imposed by the use of `eval()`. Nikiforakis *et al.* [23] analyzed script inclusions on Alexa top 10K websites and revealed four vulnerabilities that could be exploited to attack popular websites. Lauinger *et al.* [17] showed many websites included outdated or vulnerable libraries, and popular libraries (*e.g.*, jQuery) could be included for multiple times. In this work, we focus on the global identifier conflicts rather than the trust relationship between scripts. Except for function definition conflicts that could result from duplicate inclusions, JSObserver can also detect other categories of conflicts.

Dynamic Analysis of JavaScript. Many static analysis approaches (*e.g.*, [1–3]) fail to reason about the runtime behaviors of JavaScript code. Therefore, prior works have studied the dynamic analysis of JavaScript. Gong *et al.* [10] proposed DLint to dynamically detect the violations of several coding quality rules at run time. In [8], the authors combined static and dynamic analysis to detect suspicious JavaScript code, *e.g.*, unusually long functions. In contrast, our analysis focus on statement-level conflicts, especially the conflicting writes to memory locations in different scripts. In [16], the authors allowed forced execution of JavaScript code to explore all the possible paths and revealed malware behaviors. Other analysis includes data race detection [14, 22, 29], determinacy analysis [32], JavaScript performance profiling [9], concurrency error detection [13] and crash path computation [18]. These techniques are orthogonal to JSObserver, which focuses on JavaScript global identifier conflicts.

7 CONCLUSION

We have investigated a form of JavaScript code integrity problem—the JavaScript global identifier conflict problem on the Web, with an analysis framework developed based on the Chromium browser. We collected data from the main pages of the Alexa top 100K websites and detected three categories of conflicts. We demonstrated that many websites were affected by identifier conflicts. In particular, we detected 145,918 conflicts on 31,615 popular websites. It is even alarming that third-party scripts could compromise the integrity of first-party code and cookies because of the privilege of accessing the same global namespace. Our research shows that JavaScript global identifier conflict is an emerging threat to both the web users and the integrity of web applications, and highlights the need to isolate JavaScript code from different organizations.

REFERENCES

[1] 2015. The Closure Linter enforces the guidelines set by Google. https://code.google.com/p/closure-linter/.

[2] 2018. JSLint. http://www.jslint.com/.

[3] 2019. ESLint. http://eslint.org/.

[4] 2020. Cookie Control by Civic - GDPR Cookie Compliance Solution. https://www.civicuk.com/cookie-control.

[5] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2016. Content security problems? evaluating the effectiveness of content security policy in the wild. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1365–1375.

[6] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2018. Semantics-based analysis of content security policy deployment. *ACM Transactions on the Web (TWEB)* 12, 2 (2018), 1–36.

[7] Facebook. 2019. Like Button for the Web. https://developers.facebook.com/docs/plugins/like-button.

[8] Amin Milani Fard and Ali Mesbah. 2013. Jsnose: Detecting javascript code smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 116–125.

[9] Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: pinpointing JIT-unfriendly JavaScript code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 357–368.

[10] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: dynamically checking bad coding practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 94–105.

[11] Brian Hackett and Shu-yu Guo. 2012. Fast and precise hybrid type inference for JavaScript. *ACM SIGPLAN Notices* 47, 6 (2012), 239–250.

[12] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Lake Buena Vista, FL.

[13] Shin Hong, Yongbae Park, and Moonzoo Kim. 2014. Detecting concurrency errors in client-side java script web applications. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 61–70.

[14] Casper S Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. 2015. Stateless model checking of event-driven applications. *ACM SIGPLAN Notices* 50, 10 (2015), 57–73.

[15] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *International Static Analysis Symposium*. Springer, 238–255.

[16] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. J-force: Forced execution on javascript. In *Proceedings of the 26th international conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 897–906.

[17] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2018. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. *arXiv preprint arXiv:1811.00918* (2018).

[18] Magnus Madsen, Frank Tip, Esben Andreasen, Koushik Sen, and Anders Møller. 2016. Feedback-directed instrumentation for deployed JavaScript applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 899–910.

[19] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. Montréal, Canada.

[20] MDN web docs. 2019. DOM onevent handlers. https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Event_handlers.

[21] Mozilla. 2016. Subresource Integrity. https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity.

[22] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2015. Detecting JavaScript races that matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 381–392.

[23] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*. Raleigh, NC.

[24] Norris Boyd et al. 2019. Rhino: JavaScript for Java. http://www.mozilla.org/rhino/.

[25] Jibesh Patra, Pooja N Dixit, and Michael Pradel. 2018. Conflictjs: finding and understanding conflicts between javascript libraries. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. Gothenburg, Sweden.

[26] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. Florence, Italy.

[27] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G Zorn. 2010. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. *WebApps* 10 (2010), 3–3.

[28] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from big code. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 111–124.

[29] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective race detection for event-driven programs. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 151–166.

[30] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The eval that men do. In *European Conference on Object-Oriented Programming*. Springer, 52–78.

[31] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *ACM Sigplan Notices*, Vol. 45. ACM, 1–12.

[32] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic determinacy analysis. In *Acm Sigplan Notices*, Vol. 48. ACM, 165–174.

[33] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 18th European Software Engineering Conference (ESEC) / 21st ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Saint Petersburg, Russia.

[34] W3C. 2018. Content Security Policy Level 3. https://www.w3.org/TR/CSP3/.

[35] Chuan Yue and Haining Wang. 2013. A measurement study of insecure javascript practices on the web. *ACM Transactions on the Web (TWEB)* 7, 2 (2013), 7.