




Preventing Privacy-Violating Information Flows in JavaScript Applications Using Dynamic Labelling

Sandip Ghosal¹(✉)  and R. K. Shyamasundar²

¹ Department of Information Technology, Uppsala University, Uppsala, Sweden
sandipsmit@gmail.com

² Department of Computer Science and Engineering,
Indian Institute of Technology Bombay, Mumbai, India

Abstract. Web browser-based applications deal with humongous user information using applications of web scripts. In particular, JavaScript applications access information through built-in browser APIs that dynamically load remote scripts and execute with the same privilege as that of the applications – usually referred to as mashup model. Unfortunately, this allows malicious JavaScripts to manipulate the given browser functionalities leading to various web attacks violating users' privacy. Moreover, with the rapid growth of e-commerce sectors, malicious scripts pose a significant challenge to digital transactions. In this paper, we propose an approach that prevents various web-based attacks such as code injection, cross-site scripting (XSS). The approach adopts a *Dynamic Labelling* algorithm that generates information flow security policies automatically for local variables in JavaScript based on the given policies for sensitive variables. Unlike existing solutions that are too conservative primarily due to the generic flow policies leading to false-alarms, our solution leads to realizing conditions as to when a script accepts the parameters returned by a dynamic script; thus enabling us to build an automatic platform for preventing information flows to malicious scripts without explicit characterization by programmers or users.

1 Introduction

JavaScript being the most popular scripting language [2], has been an integral part of modern web applications for quite a long time. The popularity continues to grow due to a number of factors such as the flexibility to pull the necessary libraries at run-time from remotely diverse sources, delegate source code over the network to different receivers across the web for execution, and JavaScript APIs that allow sharing the page state with dynamically loaded scripts. Since confidentiality and integrity are major concerns for cross-domain sharing of code and data, modern web browsers enforce the *same-origin policy* (SOP) for isolating content and scripts from different domains. SOP specifies that data of an

S. Ghosal—Work done while the author was at Indian Institute of Technology Bombay.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022

V. R. Badarla et al. (Eds.): ICISS 2022, LNCS 13784, pp. 202–219, 2022.

https://doi.org/10.1007/978-3-031-23690-7_12

origin shall be accessible only by the code from the same origin [8,45,49]. However, there are many occasions where browsers can bypass SOP by inspecting properties of cross-origin images, frames, and windows which make the applications vulnerable to attacks such as cross-site scripting (XSS), cross-site request forgery (CSRF), and privacy leaks [3,4]. Problem is quite pronounced with the folklore “web-mashups”, a regular phenomenon these days thanks to “ads” on nearly all websites. Moreover, frequent changes in the script, code reuse and dependencies on unequally trusted third-party libraries [33] pose a significant threat to security and privacy.

Problem Description:

We illustrate the problem through the attack (Cf. [14]) shown in Figs. 1 and 2. For a given website, let a part of the script that does not load external resources be called *context*. On the other hand, the part that dynamically loads resources located remotely is referred to as the *hole*. Consider the script shown in Fig. 1, where the dynamic part loads external resources from *B*, that eventually re-directs the login parameters to an evil site *C* instead of the original destination *D* (Fig. 2). The solution proposed in [14] follows a mechanism called *staged information flow* (SIF) that prevents flows from sensitive variable to hole or

```

1  /* the context part of the script */
2  <script type="javascript">
3  var initSettings = function(s, i){
4      baseUrl = s;
5      version = i;
6  }

8  /* set original destination as D */
9  initSettings(D,1.0);

11 var login = function(){
12     var pwd = PasswordTextBox.value;
13     var uname = UsernameTextBox.value;
14     var params = "u=" + user + "&p=" + pwd;
15     post(baseUrl, params);
16 }
17 </script>
18 <text id="UsernameTextBox">
19 <text id="PasswordTextBox">
20 <button id="ButtonLogin" onclick="login()">

22 /* a hole pulls resources from B */
23 <div id="advertise">
24 <script src="B/ad.js"/>
25 </div>

```

Fig. 1. Script loads an external resources from *B*

```

1 <script type="javascript">
2 var z1 = C;
3 var z2 = 1.0;
4 /* destination redirected to C */
5 initSettings(z1, z2);
6 </script>

```

Fig. 2. Script in B redirects the destination to an evil site C

vice-versa. As per [14], a confidentiality policy is given as (A, \bullet) read as information in A cannot flow to a *hole*. However, it may be observed that blocking information from flowing to a hole beforehand may not be a good idea – makes the system too restrictive. For example, a script might be loaded to calculate the tax amount in different currencies, where the system demands information to flow to a hole but not a nested hole – this cannot be allowed through SIF.

Yang *et al.* [48] argue that information flow control (IFC) subsumes SOP and could provide a stronger notion of confidentiality and integrity. Taking inspiration from [48], in this paper, we propose an IFC-based solution that does not require explicit characterization of a *hole* by the user; however, the security (or insecurity) can be evaluated automatically through a recently proposed compile-time *Dynamic Labelling* (DL) algorithm [21, 22]. The algorithm, with respect to a given set of fixed information flow policies for global sensitive objects, dynamically generates security policies for local objects of a program. Failing to generate a security policy for a local object(s) at any program point, the algorithm indicates a potential flow policy violation. We adopt the mechanism of DL algorithm in the context of JavaScript applications to prevent information from flowing to a malicious script or vice-versa.

In summary, the paper makes the following contributions:

- Highlights some of the existing problems for enforcing flow security in JavaScript applications;
- Describes how an extension of IFC-based program certification approach, i.e. *Dynamic Labelling* algorithm, does provide potential solutions to the problems; and
- Propose an algorithm for JavaScript application based on the Dynamic Labelling approach that provides an automatic solution for preventing privacy-violating information flows.

Structure of the Paper: Section 2 provides a brief background about the security label-based information flow control, DL algorithm, and a lattice-based security model, i.e. *Readers-Writers Flow Model*. Section 3 describes the pressing problems for enforcing information flow security in JavaScript applications. Further, we discuss how an extension of DL algorithm could solve these problems. Section 4 describes our proposed approach using DL algorithm that provides an automatic solution for preventing privacy-violating information flows in

JavaScript applications. Section 5 describes the related work, followed by Sect. 6 summarizing the contribution along with the future work.

2 Background

In this section, we briefly introduce concepts of IFC that are necessary to understand the solution this paper presents. We shall start with the notion of security labels or policies or classes (used interchangeably) and the definition of *information flow policy*. Next, we introduce DL algorithm – the backbone of this paper, and a noble security model, i.e. *Readers-Writers Flow Model* (RWFM), required for governing flow transitions and in the event of *declassification* [36, 52].

2.1 A Brief Introduction to IFC

Usually, IFC systems govern information propagation by associating a security label with every subject (stakeholder of a system) and object (e.g., variable, file, register) – commonly referred to as *security labeling*. Security labels are used to specify who may read (confidentiality policy) or write data (integrity policy). For example, an object labeled as $\langle \text{readers} : \{A, B\}, \text{writers} : \{B\} \rangle$ can only be read by A, B and modified by B . In general, the set of security labels are partially ordered over a binary relation \leq read *can-flow-to*: for any objects x, y labeled as $\underline{x}, \underline{y}$ respectively, if $\underline{x} \leq \underline{y}$ then the policy encoded by \underline{x} is no more restrictive than that of \underline{y} . In other words, security policy in \underline{x} is upheld by the policy \underline{y} , hence information can flow in the direction $x \rightarrow y$.

Following the semantics of information flow, we can fairly assume two binary class combining operators, *join* (\oplus) and *meet* (\otimes), that evaluate *least upper bound* (LUB) and *greatest lower bound* (GLB) of two security labels respectively. Next we shall define a universally bounded security lattice w.r.t a partially ordered set (\mathcal{L}, \leq) , join, and meet operations.

Definition 1 (Information-Flow Lattice [16]). *A given partially ordered set (\mathcal{L}, \leq) together with a least upper bound (LUB) operator $\oplus : \mathcal{L} \times \mathcal{L} \rightarrow \mathcal{L}$ and greatest lower bound (GLB) operator $\otimes : \mathcal{L} \times \mathcal{L} \rightarrow \mathcal{L}$, defines a universally bounded lattice $(\mathcal{L}, \leq, \oplus, \otimes)$ such that there exist a lower bound $\perp \in \mathcal{L}$ where $\perp \leq l$, and upper bound $\top \in \mathcal{L}$ where $l \leq \top$ for all $l \in \mathcal{L}$.*

Using the lattice structure, an information flow policy (IFP) is defined as follows: if there is information flow from x to y then the flow is secure iff $\lambda(x) \leq \lambda(y)$, where λ is a labeling function that maps subjects and objects of a program to respective security labels from the lattice. Note that a two-point lattice could have only labels *high* and *low*, where information shall only flow in the direction *low* \rightarrow *high*. A program is certified for IFP if there are no violations of the policy during program execution [17].

2.2 Dynamic Labelling (DL) Algorithm [21,22]

The DL algorithm certifies a program for flow security by enforcing IFP at each program point with respect to a given set of fixed security labels of sensitive variables. The algorithm accepts the following input parameters: (i) a program statement S ; (ii) a labeling function λ that maps a subject or object to its respective security label in the lattice; and (iii) a clearance label cl – the highest label that the executing subject can achieve. If the statement is flow-secure the algorithm outputs a new labeling function mapping each local variable in S to their respective final label in the lattice. Otherwise, it throws an error indicating a possible IFP violation. The algorithm follows a combination of static and dynamic (or hybrid) binding of security labels with subjects and objects: a set of sensitive variables referred to as global are given fixed or static security labels, whereas the labels of intermediate or local variables, including the program counter (pc), dynamically change as the value changes. We refer to the work [22] for the properties and formal analysis of DL algorithm.

2.3 Readers-Writers Flow Model (RWFM) [32]

We borrow a lattice-based information flow control model, i.e., Readers-Writers Flow Model (RWFM) for labeling subjects and objects and governing information flows in a program. In RWFM, a subject or principal is a string representation of a source of authority such as user, process, also called as *active* agent of a program responsible for information flow. On the other hand, objects are passive agents such as variables, files used for storing information.

A RWFM label l of a subject or object is a three-tuple (s, R, W) ($s, R, W \in$ set of principals P), where s represent the owner of the information and policy, R denote the set of subjects (*readers*) allowed to read the information, and W refer to the set of subjects (*writers*) who have influenced the information so far. The readers and writers set, respectively, specify the confidentiality and integrity policy associated with the information. Information from a source with a RWFM label L_1 can flow to an endpoint having RWFM label L_2 ($L_1 \leq L_2$) if it does not violate the confidentiality and integrity policies already imposed by L_1 . The definition of *can-flow-to* is given below:

Definition 1 (Can-flow-to relation (\leq)). *Given any two RWFM labels $L_1 = (s_1, R_1, W_1)$ and $L_2 = (s_2, R_2, W_2)$, the can-flow-to relation is defined as:*

$$\frac{R_1 \supseteq R_2 \quad W_1 \subseteq W_2}{L_1 \leq L_2}$$

The join (\oplus) and meet (\otimes) of any two RWFM labels $L_1 = (s_1, R_1, W_1)$ and $L_2 = (s_2, R_2, W_2)$ are respectively defined as

$$L_1 \oplus L_2 = (-, R_1 \cap R_2, W_1 \cup W_2), \quad L_1 \otimes L_2 = (-, R_1 \cup R_2, W_1 \cap W_2)$$

Then the set of RWFM labels $SC = P \times 2^P \times 2^P$ forms a bounded lattice ($SC, \leq, \oplus, \otimes, \top, \perp$), where (SC, \leq) is a partially ordered set and $\top = (-, \emptyset, P)$, and $\perp = (-, P, \emptyset)$ are respectively the maximum and minimum elements.

Definition 2 (Declassification in RWFm). *The declassification of an object o from its current label (s_2, R_2, W_2) to (s_3, R_3, W_3) as performed by the subject s with label (s_1, R_1, W_1) is defined as*

$$\frac{s \in R_2 \quad s_1 = s_2 = s_3 \quad R_1 = R_2 \quad W_1 = W_2 = W_3 \quad R_2 \subseteq R_3 \quad (W_1 = \{s_1\} \vee (R_3 - R_2 \subseteq W_2))}{(s_2, R_2, W_2) \text{ may be declassified to } (s_3, R_3, W_3)}$$

This says, the owner of an object can declassify the content to a subject(s) only if the owner is the sole writer of the information or that subject(s) had influenced the information earlier.

3 Security Challenges and Our Approach

In this section, we outline some of the security challenges in JavaScript and illustrate how a natural extension of the DL algorithm can overcome the challenges by enforcing information flow security in JavaScript programs. The following constructs constitute the core of our JavaScript language: assignment, selection, iteration, sequence, `eval`, and functions. A subset of ECMA-262 [1] standard defines the syntax and semantics of our JavaScript constructs. In addition, we provide a solution to declassification in JavaScript applications that could arise due to decentralized labeling [35] followed by the modern security lattice models.

Before delving into the challenges, we shall briefly revisit the three information flow channels, i.e. *explicit*, *implicit*, and *covert*, responsible for leaking information [42]. An assignment $\ell = h$ is the most prevalent example of information leak through explicit channel where sensitive (or *high*) information in h is directly copied into a public (or *low*) variable ℓ . An implicit channel occurs in a conditional statement such as `if...then...else` when a branch is selected based on a condition involving sensitive values. Finally, covert channels refer to all other side channels where information leaks may occur depending on the program (non)termination, power consumption, or execution time.

In the presence of any of the above information channels, the program violates the non-interference property [23, 46]. Usually, enforcement of the global policy of non-interference ensures information shall flow only in the upward direction in the lattice ($low \rightarrow high$). While there has been impressive progress in both the static [12, 24, 37] and dynamic [6, 50, 53] flow security analyses, however, their applications for enforcing non-interference in JavaScript are limited due to the reasons discussed in the following.

3.1 Flow Sensitivity

Since JavaScript is a dynamically typed language, the type of variables and fields changes as per the information flow during execution, making the type system information flow sensitive. However, classical dynamic flow sensitive analyses [20, 26, 40] might allow the following program to copy sensitive information (0 or 1) in h (labeled *high*) to public variable ℓ (labeled *low*) via temporary variable t without upgrading the security label of ℓ .

$$\ell = 0; t = 1; \text{ if } (h == 0) t = 0; \text{ if } (t! = 0) \ell = 1;$$

Our Approach: DL algorithm follows a hybrid labeling approach where the labels of global variables h and ℓ are static, but the label of t is dynamic. As it encounters an implicit flow from h to t conditioned on h , eventually updates the label of t to *high*. Therefore, in the following branch, the algorithm can identify a violation of non-interference caused by the implicit flow $t \rightarrow \ell$.

Table 1. DL algorithm for JavaScript assignment statements

Statement	Flow constraints
<code>baseUrl = s</code>	$\lambda(s) \oplus \lambda(pc) \leq \lambda(baseUrl)$
<code>var pwd = PasswordTextBox.value</code>	$\lambda(pwd) = \lambda(PasswordTextBox.value) \oplus \lambda(pc)$

For the example shown in Fig. 1, consider objects *baseUrl* and *pwd* are global and local respectively. Then Table 1 shows the flow check and label computation performed by DL algorithm for the assignment statements at lines 4 and 12. Since the variable *baseUrl* has a fixed security label, the algorithm only checks if the label of *s* can flow to *baseUrl*, otherwise raises an error indicating an incident of possible flow policy violation. On the other hand, the algorithm dynamically updates the label of local variable *pwd* to accommodate information flow from the object *PasswordTextBox.value*. Therefore, the algorithm follows static labeling when the target is a global variable; whereas, it uses dynamic labeling by updating labels when the target is a local variable. Thus, DL algorithm is flow sensitive for local variables; in addition, it overcomes issues of overapproximation of security labels often encountered in static analyses that could lead to false alarms for security certification. An approach called *no-sensitive-upgrade* [50] overcomes the shortcomings of dynamic analyses for information leaks through implicit channels. The mechanism only allows assignment to variables that have a security label at least as high as the label of the predicate that governs the assignment. Thus, it avoids upgrading variables of only the branch in execution. While the DL algorithm is in line with *no-sensitive-upgrade* method, note that the latter is *termination-insensitive* [5] – could leak information based on (non)termination of the program.

3.2 Termination Sensitivity

The existing IFC mechanisms enforce an imprecise notion of information flow security when it comes to *termination-sensitive* non-interference [5, 47]. Under this class of non-interference, information leak takes place depending on whether the program has terminated or not. Askarov *et al.* [5] argued that such an imperfection of flow analysis is the price to pay for having a security condition that is relatively liberal (e.g., allowing while loops whose termination may depend on the value of a secret) and easy to check. According to the authors, in the presence of output to a public channel, the price is higher than just “one-bit” often

claimed informally in the literature, and effectively such programs can leak all of their secrets. In the case of JavaScript, the presence of a termination leak could be fatal; e.g., consider the program shown in Fig. 3, where h is a high-security variable storing sensitive information and $img.url$ is an object which stores the path to an image such that $\lambda(h) \not\leq \lambda(img.url)$. Note that the image is displayed only if the value in h is 1; hence the information is leaked depending on the (non)termination of the loop.

```
1 while (h == 0){ }
2 img.url = 'http://abc.com/img.jpg';
```

Fig. 3. An example of information leak through a termination channel in JavaScript

Handling Termination Sensitivity in Our Approach: DL algorithm is termination-sensitive: it captures information leaks through termination channels (covert channel) that could arise due to (non)termination of the loop statements. DL algorithm tracks the pc label and updates it monotonically as it reads a new variable. This enables the algorithm to capture forward information flow that implicitly leaks information from loop predicate to the following statement. A recurring backward information flow could also arise due to multiple loop iterations, which may update the label of local variables in each iteration and could eventually violate IFP. The DL algorithm continues to iterate the loop until it computes the highest possible label of the local variables in the loop. As it can be shown that the highest labels are achieved within three iterations only, DL algorithm terminates the loop after a maximum of three unrolls [22]. Note that the loop might not terminate during actual execution in run time. In this process, the algorithm enforces IFP in the presence of backward information flow. In the above example, the algorithm captures the insecurity in the following way: reads variable h in the predicate; updates the label of pc to the equivalent of h ; terminates the while statement after iterating the loop once as it does not have any local variable in the body; moves forward to the next statement; realizes the insecurity as it fails to satisfy the flow constraint $\lambda(pc) \leq \lambda(img.url)$.

Devriese and Piessens [18] proposed an approach called *Secure Multi-Execution* (SME) that enforces termination-sensitive non-interference by executing multiple instances of a program concurrently, once for each security label in the lattice, applying special rules for I/O operations that prevent the information flow from high to low-security variables. An instance of a program executed with the security label ℓ can only write to outputs with security label ℓ and can read from the inputs with security labels $\ell' \leq \ell$. A default input statement replaces the inputs with higher security labels. SME intercepts executions of inputs and outputs that violate the above I/O rules. An execution under SME at a given security label can only produce output with the same label and cannot see inputs from a higher security label. Therefore, outputs produced under SME could not possibly depend on higher inputs.

Unlike DL algorithm, SME framework is built upon a two-point lattice, where the security labels (*high* or *low*) for each program instance shall be given a priori. An extension to a general n -point lattice would require executing n program instances concurrently, which could be limited by the program design, system architecture, and scheduling strategy. Besides, restrictions on I/O operations could limit the application of SME for concurrent programs that exchange information using shared variables. Although the approach adheres to *no-sensitive-upgrade* and enforces non-interference as it prevents secret input from influencing the public output [6], in practice, the information flow usually occurs in the opposite direction, where high-security inputs always influence low-security outputs [41]. Having said that, we arrive at the notion of declassification, a common phenomenon for developing *multi-level security* (MLS) systems. However, how one would define declassification using SME remains unclear. We discuss declassification in JavaScript applications in the sequel.

3.3 Eval Statement

An `eval` instruction in JavaScript could parse and execute a string argument as code at run-time. Thus, `eval` statement could execute a malicious statement with the same privilege as the caller application. Since the string may not be available to a static analyzer, it poses a significant challenge to static flow analyses.

Eval Statement Treatment in Our Approach: Taking inspiration from the work by Jang *et al.* [28, 29], we could follow a rewriting-based source-to-source transformation to generate an intermediate representation of JavaScript code for `eval` statements. For rewriting, we can fairly assume an approximate string-matching function \mathcal{A} that maps a given input string s to command(s) c matched with the JavaScript syntax. Whereas the function \mathcal{A}_{Rev} performs the reverse for a given program statement(s). Then as shown in Fig. 4, the rewriting function \mathcal{R} could interpose either of the following two treatments to the string argument: (i) replace the `eval` statement with the argument itself, or (ii) a function call within the scope of caller application where the function performs the operations that would have executed by the `eval`.

Next, DL algorithm shall generate security labeling for intermediate variables of transformed JavaScript source code. In case of the latter approach, the algorithm performs flow analyses similar to a function call [22]. Note that the clearance label cl for caller application would then be treated as an upper bound for evaluating the function body. Therefore, the execution of an `eval` statement is flow-safe as long as the function call in the transformed JavaScript code does not invalidate the *can-flow-to* relation with the static labels of sensitive variables in the caller application.

$$\begin{array}{ll}
S' := S[T(\text{eval}(s)) \mapsto \mathcal{A}(s)]; & S' := \text{function } f : \mathcal{A}(s) \# \# S; \\
\text{output } S'; & s' := \mathcal{A}_{\text{Rev}}(f()); \\
& S'' := S'[T(\text{eval}(s)) \mapsto \text{eval}(s')]; \\
& \text{output } S'';
\end{array}$$

Fig. 4. Steps performed by $\mathcal{R}(S)$ for a JavaScript S when the function T transforms **eval** through replacement (left); and function call (right). The binary operator ‘ $\# \#$ ’ produces a sequence of program statements.

3.4 Declassification

Often enforcement of the global policy of non-interference is too restrictive to design real systems. In practice, system design often demands information to flow from *high* to *low* as one can intuitively see in the classic *password example*. For this purpose, the notion of declassification [27, 31, 36, 43, 52] has been used extensively under rigid conditions so that confidentiality is not violated to the detriment of the usage. Declassification eventually allows more subjects to be readers of the information. However, the addition of readers needs to be genuinely robust as it may otherwise reduce to pure discretionary access control that has severe consequences in a decentralized model.

Motivated by the earlier development of SME [18], Austin & Flanagan [7] proposed the notion of *faceted value*, a pair of raw values (facets) containing *low* and *high* information respectively. In this work, the authors introduce a mechanism called *facet declassification* where information shifts from one facet to another only if the control path has not been influenced by an untrusted label. Although the work follows the definition of *robust declassification* [51], it does not define the drop in confidentiality, i.e., who could be the potential reader of the declassified data, hence it could boil down to discretionary access control. Bauer *et al.* [9] proposed a run-time monitor for web browsers introducing an information-flow label written as a tuple (S, I, D) , where S , I , and D denote the secrecy, integrity, and declassification labels respectively. Information can flow from a sender to a receiver, labeled as $(S_1, I_1, \{\})$ and $(S_2, I_2, \{\})$ respectively, only if $S_1 \subseteq S_2$ and $I_1 \supseteq I_2$. The declassification label is often useful to circumvent the above constraints, which otherwise would have failed due to secrecy and integrity tags. However, the declassification could disclose sensitive data to any arbitrary entity if the label is not given judiciously. Moreover, the approach does not follow the notion of *robust declassification* as the mechanism of declassification is independent of secrecy or integrity labels.

Our Approach to Declassification using RWFM: Consider a password manager extension P that stores the username and password for websites (or server or entity) A , B , and C each time a user performs a new login and enables the browser to auto-fill the same for every subsequent login to that respective website. We apply DL algorithm in tandem with RWFM label specification and governing policies for flow transitions and declassification. Firstly, we shall provide static labels for global variables such as username or password for website

A as $\lambda(A) = (A, \{P, A\}, \{A\})$, read as A is the owner, P and A are readers and only A has written the information so far. Secondly, the initial dynamic label for the password manager extension is given as $\lambda(P) = (S, \{*\}, \{\})$, meaning S is the owner (could be the user or browser) of the password storage, $'*'$ denotes anybody can read the information (public), and an empty writer set represents that nobody has written the information so far. Next, the label of P changes dynamically according to flow transitions. The label changes for P are shown in Table 2 where the column “Transitions” represents the information flow direction, i.e. $A \rightarrow P$ when password manager reads username and password entered for the website A , and $P \xrightarrow{d} A$ when P declassifies information to A . Note that the reader set of P becomes more restrictive as it collects information from A , B , and C . As the label of P obtains the highest possible label in the lattice, information cannot flow from P to the websites A , B , or C for the sake of auto-filling the username and password fields on subsequent visits. Therefore, the label of P needs to be declassified to the respective website. During the next login to website A , RWFM declassification eventually includes A into the reader set as A has influenced the information earlier; in other words, A is present in the writer set of P . Unlike the earlier approaches, RWFM declassification clearly defines the drop in confidentiality (readers) policy and prevents declassifying sensitive data to an entity that has not influenced the data in preceding flow transitions. In addition, the approach also automatically generates dynamic labels for local variables, therefore not requiring explicitly providing labels for each entity.

Table 2. Changes in RWFM labels corresponding to information flows

Transitions	Source label	Destination label	Changed label
$A \rightarrow P$	$\lambda(A) = (A, \{P, A\}, \{A\})$	$\lambda(P) = (S, \{*\}, \{\})$	$\lambda(P) = (S, \{P, A\}, \{A\})$
$B \rightarrow P$	$\lambda(B) = (B, \{P, B\}, \{B\})$	$\lambda(P) = (S, \{P, A\}, \{A\})$	$\lambda(P) = (S, \{P\}, \{A, B\})$
$C \rightarrow P$	$\lambda(C) = (C, \{P, C\}, \{C\})$	$\lambda(P) = (S, \{P\}, \{A, B\})$	$\lambda(P) = (S, \{P\}, \{A, B, C\})$
$P \xrightarrow{d} A$	$\lambda(P) = (S, \{P\}, \{A, B, C\})$	$(S, \{P, A\}, \{A, B, C\})$	

There are two possibilities for the placement of declassification construct: (i) have an assertion that ensures declassification explicitly, or (ii) perform declassification implicitly at the JavaScript function return. In this work, we consider the former option as implicit declassification or automatic placement of the construct could lead to security risk – we leave the discussion for our future work.

4 Solution for Preventing Privacy-Violating Flows

We extend the DL algorithm for identifying privacy-violating information flows in JavaScript programs. DL algorithm generates dynamic labels for local objects of a JavaScript program for a given set of user-defined fixed labels of global objects (variables interact with the outside world). Possibility (or otherwise) of labeling the program objects leads to certification (or otherwise) of the program;

the same could be used in the execution monitor for checking flow security at run-time as discussed for **eval** statement in the previous section.

Let W be a website, S and H denote the static and dynamic part of the script respectively, λ be a given labeling function, and cl is the clearance label. The DL algorithm for identifying privacy-violating information flow is shown in Table 3. Note that the information flow between the static and dynamic part of the script is administered by the binary relation *can-flow-to* of the security lattice.

Table 3. DL Algorithm for JavaScript application

DL(W, λ, cl) ::
1. $\lambda_1 = \text{DL}(S, \lambda, cl)$
2. WHILE there exists H then
3. load H
4. $\lambda_2 = \text{DL}(H, \lambda, cl)$
5. IF there is a flow from $S(param)$ to $H(var)$
6. Check $\lambda_1(param) \leq \lambda_2(var)$
7. IF there is a flow from $H(var)$ to $S(param)$
8. Check $\lambda_2(var) \leq \lambda_1(param)$
9. IF H contains H' then
10. Go to step 2
11. ELSE Exit

Table 4 demonstrates the application of DL algorithm for the example shown in Fig. 1. Consider variables *baseUrl*, *version*, *PasswordTextBox.value*, and *UsernameTextBox.value* as global objects, whereas *pwd*, *uname*, and *params* are local objects. An initial labeling function λ maps the global objects to the corresponding given static security label and local objects to \perp . Next, for the given initial mapping λ , our proposed algorithm performs the following tasks: (i) computes the final labels of local variables in the context part, and thus, obtains a new labeling function λ_1 ; (ii) loads the hole part of the JavaScript; (iii) obtains a labeling function λ_2 that maps each local variables of the hole to their respective final security labels; and (iv) checks if the required flow constraints hold whenever there are information flows from the hole to the context or vice-versa. Thus, the algorithm allows the context part of the script to accept the parameters of a hole only if the required flow constraints at the line **initSettings**($z1, z2$) hold good.

Note that unlike SIF [14], our framework built upon the proposed algorithm allows the information to flow to/from a hole as long as it does not violate the flow constraints. Thus, our approach could lead to the development of a flow-secure web browser by modifying the browser core or implementing an OS-

agnostic browser extension that would enable us to specify and enforce security policies and analyze information flows in JavaScript against the given policies.

5 Related Work

In this section, we briefly describe previous IFC-based solutions for preventing privacy-violating leaks in JavaScript applications.

Table 4. A solution to example in Fig. 1 using DL algorithm

	Script	Labeling
Context	<pre> var initSettings = function(s, i){ baseUrl = s; version = i; } initSettings(D, 1.0); /*initial destination D/* var login = function(){ var pwd = PasswordTextBox.value; var uname = UsernameTextBox.value; var params = "u = " + user + "&p = " + pwd; post(baseUrl, params); } </pre>	<p>Global objects: <i>baseUrl, version,</i> <i>PasswordTextBox.value,</i> <i>UsernameTextBox.value</i></p> <p>Local Objects: <i>pwd, uname, params</i></p> <p>$\lambda_1 = DL(\text{Context}, \lambda, cl)$</p>
Hole	<pre> var z1 = C; var z2 = 1.0; initSettings(z1, z2); /*destination redirected to C/* </pre>	<p>Loads the Hole $\lambda_2 = DL(\text{Hole}, \lambda, cl)$</p> <p>Checks: $\lambda_2(z1) \leq \lambda_1(baseUrl)$ $\lambda_2(z1) \leq \lambda_1(version)$</p>

Most of the earlier approaches to IFC in JavaScript are based on dynamic information flow control (DIFC) [19, 25, 28, 38]. DIFC attaches *high* or *low* security labels with sensitive or insensitive values respectively and propagates the labels during program execution. Vogt *et al.* [38] proposed a tainting-based approach that marks sensitive data, and the sensitivity is propagated when the data is accessed by scripts running in the web browsers. Dhawan and Ganapathy [19] proposed a similar approach where a label is associated with each in-memory JavaScript object. The label determines if an object contains sensitive information. The labels are propagated as objects are modified by the JavaScript engine and passed between browser subsystems. The technique raises an alert when a

sensitive object is accessed in unsafe way. Jang *et al.* [28] proposed a dynamic code rewriting-based mechanism that enables injecting taints depending on different privacy-violating flows and eventually propagated and blocked. Hedin *et al.* [25] first implemented DIFC in the form of a JavaScript interpreter called *JSFlow* for fine-grained tracking of information flow in large JavaScript applications. However, the above approaches fail to realize termination leaks due to the program's non-termination.

Some variant of inline reference monitor (IRM) performs dynamic inline taint tracking that does not require browser modification but instruments the code or existing JavaScript engine – the interpreter or just-in-time (JIT) compiler [11, 13]. The mechanisms must translate JavaScript to prevent security risks due to dynamic features. Although some of the approaches incur moderate performance overhead, unlike our approach, the non-interference is termination-insensitive.

S. Just *et al.* [30] first introduced a hybrid mechanism that dynamically tracks explicit information flows while tracking intra- and inter-procedural information flows using static analysis. Following the hybrid information flow analysis, Bedford *et al.* [10] proposed a flow-sensitive inline monitoring that uses an oracle to determine loop termination behaviour [34]. While the former approach does not guarantee termination-sensitive non-interference, the notion of declassification is yet to be defined in the latter.

Devriese and Piessens [18] proposed *Secure Multi-Execution* (SME) that executes multiple instances of a JavaScript program concurrently, once for each security label in the lattice, applying special rules for I/O operations that prevent the information flow from high to low-security variables. However, in practice, the information flow usually occurs in the opposite direction, where high-security inputs always influence low-security outputs [41]. Particularly, in cryptographic operations, files are encrypted, sanitized, declassified, and sent over low-level public network. But the implementation of declassification [36, 52] in SME and its subsequent developments [15, 39] is majorly overlooked.

Recent development claims that the cases of information leaks due to implicit flows are insignificant in web applications, but tracking implicit flows is expensive and incurs performance overhead [44]. Comparatively, a lightweight taint analysis could be sufficient to track insecurity due to explicit flows. Nonetheless, ignoring implicit flows could have a severe security risk; therefore, our approach stands out in this context that could prevent leaks due to implicit, explicit, and termination channels. Furthermore, following RWFM specification provides a robust notion of declassification. We leave the performance evaluation of our framework and comparison with the existing implementation as our future work.

6 Conclusions and Future Work

There are two research aspects for establishing flow security in JavaScript applications. Firstly, identifying the key security challenges for enforcing flow security in JavaScript language, including declassification. Secondly, defining flow security with respect to information flows in different directions in JavaScript

applications, such as from context to context, context to hole, hole to context, and hole to hole. In this paper, we have highlighted the security challenges and provided a detailed description of our solution using an IFC-based program certification algorithm. Further, we have provided a solution to prevent cross-script privacy-violating information flows, in particular, and thus, answered the fundamental question that is *when should a script accept the parameters returned by a dynamic script?*. Further, we have proposed an all-in-one automatic solution using *Dynamic Labelling* algorithm and *RWFM* label specifications that could identify privacy-violating information flows with respect to a given set of security policies associated with sensitive objects. The solution would help coexist the cooperating scripts and encourage the “web-mashup” model - a common practice in web applications; although it involves privacy risk but relevant from business perspectives.

In future, we plan the development of a flow-secure web browser by modifying the browser core or implementing an OS-agnostic browser extension that would enable us to specify and enforce security policies and analyze information flows in JavaScript against the given policies. In addition, our future work includes extensions to handling exceptions and concurrency. In particular, we would be interested in scaling up our approach to deal with real-world applications and evaluate performance against the existing implementation.

References

1. EcmaScript 2023 language specification. <https://tc39.es/ecma262/>
2. Most popular technologies. <https://insights.stackoverflow.com/survey/2020#most-popular-technologies>
3. Cross-domain security woes. the strange zen of javascript (2005). <http://jszen.blogspot.com/2005/03/cross-domain-security-woes.html>
4. Defining safer json-p (2020). <https://json-p.org/>
5. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 333–348. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88313-5_22
6. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: Proceedings of the ACM SIGPLAN 4th Workshop on PLAS, pp. 113–124 (2009)
7. Austin, T.H., Flanagan, C.: Multiple facets for dynamic information flow. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 165–178 (2012)
8. Barth, A.: The web origin concept. Technical report (2011)
9. Bauer, L., Cai, S., Jia, L., Passaro, T., Stroucken, M., Tian, Y.: Run-time monitoring and formal analysis of information flows in chromium. In: NDSS (2015)
10. Bedford, A., Chong, S., Desharnais, J., Kozyri, E., Tawbi, N.: A progress-sensitive flow-sensitive inlined information-flow control monitor (extended version). *Comput. Secur.* **71**, 114–131 (2017)
11. Bichhawat, A., Rajani, V., Garg, D., Hammer, C.: Information Flow Control in WebKit’s JavaScript Bytecode. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 159–178. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_9

12. Broberg, N., van Delft, B., Sands, D.: Paragon for practical programming with information-flow control. In: Shan, C.-C. (ed.) APLAS 2013. LNCS, vol. 8301, pp. 217–232. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03542-0_16
13. Chudnov, A., Naumann, D.A.: Inlined information flow monitoring for javascript. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 629–643 (2015)
14. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for javascript. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 50–62 (2009)
15. De Groef, W., Devriese, D., Nikiforakis, N., Piessens, F.: Flowfox: a web browser with flexible and precise information flow control. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 748–759 (2012)
16. Denning, D.E.: A lattice model of secure information flow. CACM **19**(5), 236–243 (1976)
17. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977)
18. Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: 2010 IEEE Symposium on Security and Privacy, pp. 109–124. IEEE (2010)
19. Dhawan, M., Ganapathy, V.: Analyzing information flow in javascript-based browser extensions. In: 2009 Annual Computer Security Applications Conference, pp. 382–391. IEEE (2009)
20. Fenton, J.S.: Memoryless subsystems. Comput. J. **17**(2), 143–147 (1974)
21. Ghosal, S., Shyamasundar, R.K., Kumar, N.V.N.: Static security certification of programs via dynamic labelling. In: Proceedings of the 15th International Joint Conference on e-Business and Telecommunications, ICETE 2018 - Volume 2: SECRIPT, 26–28 July 2018, pp. 400–411 Porto, Portugal (2018)
22. Ghosal, S., Shyamasundar, R., Kumar, N.N.: Compile-time security certification of imperative programming languages. In: Obaidat, M.S. (ed.) ICETE 2018. CCIS, vol. 1118, pp. 159–182. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34866-3_8
23. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, pp. 11–11. IEEE (1982)
24. Graf, J., Hecker, M., Mohr, M.: Using joana for information flow control in java programs - a practical guide. In: Proceedings of the 6th Working Conference on Programming Languages (ATPS 2013). LNI, vol. 215, pp. 123–138. Springer, Berlin (2013)
25. Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: Jsflow: tracking information flow in javascript and its apis. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing, pp. 1663–1671 (2014)
26. Hedin, D., Sabelfeld, A.: Information-flow security for a core of javascript. In: Computer Security Foundations Symposium (CSF), 2012 IEEE 25th, pp. 3–18. IEEE (2012)
27. Hicks, B., Ahmadizadeh, K., McDaniel, P.: From languages to systems: Understanding practical application development in security-typed languages. In: 2006 22nd Annual Computer Security Applications Conference (ACSAC 2006), pp. 153–164. IEEE (2006)
28. Jang, D., Jhala, R., Lerner, S., Shacham, H.: An empirical study of privacy-violating information flows in javascript web applications. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 270–283 (2010)

29. Jang, D., Jhala, R., Lerner, S., Shacham, H.: Rewriting-based dynamic information flow for javascript. In: 17th ACM Conference on Computer and Communications Security (2010)
30. Just, S., Cleary, A., Shirley, B., Hammer, C.: Information flow analysis for javascript. In: Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients, pp. 9–18 (2011)
31. King, D., Jha, S., Jaeger, T., Jha, S., Seshia, S.A.: On automatic placement of declassifiers for information-flow security. Technical report, Technical Report NASTR-0083-2007, Network and Security Research Center (2007)
32. Kumar, N.V.N., Shyamasundar, R.: A complete generative label model for lattice-based access control models. In: Cimatti, A., Sirjani, M. (eds.) SEFM 2017. LNCS, vol. 10469, pp. 35–53. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66197-1_3
33. Mitropoulos, D., Louridas, P., Salis, V., Spinellis, D.: Time present and time past: analyzing the evolution of javascript code in the wild. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 126–137. IEEE (2019)
34. Moore, S., Askarov, A., Chong, S.: Precise enforcement of progress-sensitive security. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 881–893. ACM (2012)
35. Myers, A.C., Liskov, B.: A Decentralized Model for Information Flow Control, vol. 31. ACM (1997)
36. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Software Eng. Methodol. **9**(4), 410–442 (2000)
37. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: java information flow (2001). <http://www.cs.cornell.edu/jif>
38. Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross-site scripting prevention with dynamic data tainting and static analysis. In: Proceeding of the Network and Distributed System Security Symposium (NDSS 2007). Citeseer (2007)
39. Ngo, M., Bielova, N., Flanagan, C., Rezk, T., Russo, A., Schmitz, T.: A better facet of dynamic information flow control. In: Companion Proceedings of the The Web Conference 2018, pp. 731–739 (2018)
40. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: 2010 23rd IEEE Computer Security Foundations Symposium, pp. 186–199. IEEE (2010)
41. Ryan, P., McLean, J., Millen, J., Gligor, V.: Non-interference: who needs it? In: CSFW, p. 0237. IEEE (2001)
42. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Selected Areas Commun. **21**(1), 5–19 (2003)
43. Sabelfeld, A., Myers, A.C.: A Model for delimited information release. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS 2003. LNCS, vol. 3233, pp. 174–191. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-37621-7_9
44. Staicu, C.A., Schoepe, D., Balliu, M., Pradel, M., Sabelfeld, A.: An empirical study of information flows in real-world javascript. In: Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, pp. 45–59 (2019)
45. Van Kesteren, A., et al.: Cross-origin resource sharing. W3C Working Draft WD-cors-20100727, latest version available at < (2010). <http://www.w3.org/TR/cors> (2010)

46. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. *J. Comput. Secur.* **4**(2–3), 167–187 (1996)
47. Volpano, D., Smith, G.: Eliminating covert flows with minimum typings. In: *Proceedings 10th Computer Security Foundations Workshop*, pp. 156–168. IEEE (1997)
48. Yang, E., Stefan, D., Mitchell, J., Mazières, D., Marchenko, P., Karp, B.: Toward principled browser security. In: *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)* (2013)
49. Zalewski, M.: *Browser security handbook*. Google Code (2010)
50. Zdancewic, S.A., Myers, A.: *Programming Languages for Information Security*. Cornell University (2002)
51. Zdancewic, S.: A type system for robust declassification. *Electron. Notes Theoretical Comput. Sci.* **83**, 263–277 (2003)
52. Zdancewic, S., Myers, A.C.: Robust declassification. *CSFW.* **1**, 15–23 (2001)
53. Zheng, L., Myers, A.C.: Dynamic security labels and static information flow control. *Int. J. Inform. Secur.* **6**(2–3), 67–84 (2007)