A self-correcting information flow control model for the web-browser

Deepak Subramanian, Guillaume Hiet, Christophe Bidan

A self-correcting information flow control model for the web-browser

Deepak Subramanian, Guillaume Hiet, and Christophe Bidan

CentralSupélec
firstname.lastname@supelec.fr

Abstract. Web-browser security with emphasis on JavaScript security, is one of the important problems of the modern world. The potency of information flow control (IFC) in the context of JavaScript is quite appealing. In this paper, we propose a novel approach to help track and learn from information flows. This learnt data can subsequently be used to create a more adaptive and effective IFC model. As the information about a function augments, potential leaks are also thwarted.

1 Introduction

The state of the internet has been evolving with the constant sharing of content and functionalities between websites. Many modern technologies proposed in HTML5 have served to increase capabilities of the browser while also raising new possibilities for information leakages. For example, LocalStorage provides an efficient way for persistence in the web-browser. This could cause browser-side persistent cross-site scripting if not sanitized properly. Another example is the use of WebSockets, which provides a real-time communication channel to the server. Once the channel is open, further messages in the channel do not require re-authenticaion or cookies since the connection is already established. There is hence a clear need to monitor the browser at a variable level as JavaScript slowly dominates the application space.

Web-browser vulnerabilities have constantly been cited among the top prevalent threats as seen clearly in lists such as the OWASP Top Ten [1]. Despite the numerous safeguards proposed over the years, including some important considerations such as the "same-origin policy" and "content security policy", these threats continue exist. Moreover it is important to have a method to provide security while preserving the user experience. It is our sincere belief that Information Flow Control (IFC) could be an effective solution to this problem.

This paper is a logical extension an approach called Address Split Design (ASD) proposed by Deepak et. al. [12]. In ASD, the authors show how the use of IFC can prevent unauthorized code from accessing sensitive information thereby protecting leakage of sensitive information. ASD maintains a dummy/public value for every secret variable and uses this public value in case of unauthorized

[1] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

access. In this paper, we first propose a change to how ASD maintains runtime monitoring. This proposal could reduce the number of operations that need to be performed when a secret variable is updated. In this next part of the paper, we describe a learning-based approach to information flow control on a web-browser. After the initial execution of a program using ASD, information regarding the propagation of the secret is remembered. This information is used in a self correcting mechanism for the analysis and enforcement of security on an interpreted language such as JavaScript. The bedrock of our proposed mechanism lies in correcting any possible leak through learning rather than using rigid security guarantees. Hence, in this paper, we first start with ASD as the base model and remember all the flow propagations which are used subsequently thereby accounting for all possible information flows.

The rest of the paper is organized as follows. The section 2 provides a summary of the various related work. This is followed by a description of our model, address split design with dependency graphs (ASD-DG), along with relevant formalisms, in section 3. The details on how reinforcement learning is applied to ASD-DG and the security guarantees that can be obtained because of learning are described in section 4. There is a sum up of our approach in section 5.

2 Related work

There have been several key research since the models such as Bell and La Padula [4], and Biba [5] that have helped in shaping the field of Information Flow Control (IFC). These models established various levels of privileges and associated information to these levels. A common mechanism to achieve this is to attach security labels to information containers such as files or variables. The most simple example consists of two security labels namely, "high/sensitive" and "low/public". Information cannot flow from "High" level containers to "Low" level containers while the vice-versa is permissible. In a lattice-based design, the security lattice can contain multiple parallel or intertwined levels allowing for more complex information flow control design.

The first step in an IFC approach is to specify a policy. It consists first in defining different security labels in a lattice-based structure and then to attach these labels to the containers according to their sensitivity. Once a policy has been specified, the IFC model ensures that the execution conforms to the policy. Such an approach helps to classify information flows into legal and illegal flows. IFC models have varying mechanisms to deal with illegal information flows. These could be raising alerts, stopping execution, stopping the compilation process, modifying execution or some other customized action.

Language based IFC has been explored in great detail by label based approaches as illustrated by Sabelfeld and Myers [11]. In these approaches, the secret variables are tagged with security labels and these labels are propagated along with the information flows. In the context of IFC in JavaScript, Bielova et. al. [6] provide a substantive survey with comparative studies on techniques, types of analysis (i.e. static vs. dynamic approaches) and their formal guaran-

tees. This work has been instrumental in providing a complete picture of this research area thereby becoming a valuable stepping stone to the design of our approach.

In the field of IFC, the most important property to be satisfied by any analysis is non-interference. There are two types of non-interference based on the conditions satisfied, namely Termination-insensitive non-interference (TINI) and, Timing- and Termination-sensitive non-interference (TTSNI).

TINI [1,6,11,7] is a security guarantee where, for two terminating executions of a program with the same public input, the observable public output remains unchanged regardless of the value of the secret.

TTSNI [7,10] is a security guarantee where, for two executions of a program with the same public input, the public output, the number of execution steps and time taken to generate the public output remain unchanged regardless of the value of the secret.

There are several IFC models that have been designed for the web-browser taking into account the nature of JavaScript. Relative work in this domain has been heavily biased towards dynamic approaches. This is justifiable by the highly dynamic nature of JavaScript which increases the complexity for static approaches thereby making dynamic approaches more effective.

Label-based approaches have always been in the forefront of dynamic approaches. In the context of JavaScript, Austin and Flanagan proposed the no-sensitive-upgrade [2]. Hedin and Sabelfeld [9] proposed an IFC approach for JavaScript based on a classical label-based approach previously described by Sabelfeld and Myers [11]. Hedin and Sabelfeld show the need for dynamic approaches by describing problem of the information-flow being flow sensitive in JavaScript. This increases the need to keep track of changing labels throughout the execution which becomes tedious with pure static approaches. The difference between the two approaches is that Hedin and Sabelfeld allow some upgrade instructions before the behest of the implicit information-flow. This means that, in case an implicit flow results owing to the value of a secret variable, the public output of a public value cannot be performed under any scenario in case of the no-sensitive-upgrade. However, in case of Hedin and Sabelfeld, such a public output can be allowed if and only if there was an explicit upgrade instruction before the output statement is performed.

Both these label based approaches stop further execution of a program when they encounter a possible information leak. These approaches are useful to check if a program is adherent to TINI by default without any modifications. However, they fail to continue execution of the program if there is a possibility of a leak. There exist a few dymanic preventive enforcement mechanisms in JavaScript which are able to continue execution of the program and still prevent information leakage. These mechanisms maintain more than one copy of the variable and switch contexts of the variable based on the scenario. In case there is an unauthorized public output of a secret variable, these mechanisms use a dummy/public value instead. Secure Multi-Execution (SME), faceted approach and Address Split Design (ASD) are the models known to use this approach.

The model of SME was proposed by Devriese and Piessens [7]. In SME, the information flow across labels is segregated at the process level by providing a separate process for each level of sensitivity. Let us consider a system with two levels namely a high and a low. Such a scenario would imply that there is a dedicated process for high level computations and a dedicated process for low computations. This ensures that the memory is also safely handled since the processes themselves are isolated. The low level process is the only one that can influence public output and it can only receive public input. FlowFox is a concrete implementation of SME on the Firefox web browser by De Groef et al. [8]. However, the use of SME automatically increases the time- and space-complexity for the system.

The faceted approach that has been proposed by Austin et al. [3,2] is the chief proponent of the multi-path execution approach. The authors attempt to mimic the functionality of SME with the use of a single process. It would intuitively result in a much lower time complexity. The faceted approach attains termination-insensitive non-interference since the use of a single process cannot account for timing-sensitivity. This approach works on containing multiple copies of the variables at each juncture to mimic the values of the variables in different processes in case of SME. A faceted value is represented as <p?a_{private}:a_{public}> where p is the principal. The principal is an access control object which determines which copy of the variable should be used. If an object c were created by using two other objects a and b, each with its own principal, there would be four possible values for this object, as shown in `[FACETED APPROACH]` . This growth in

`[FACETED APPROACH]`

$$\texttt{var } a = \langle \texttt{p1?1 : 2} \rangle;\ \texttt{var } b = \langle \texttt{p2?3 : 4} \rangle;\ \texttt{var } c = a + b;$$
$$=>\ c\ =\ \langle\ \texttt{p1}\ ?\ \langle\ \texttt{p2}\ ?\ 4\ :\ 5\ \rangle\ :\ \langle\ \texttt{p2}\ ?\ 5\ :\ 6\ \rangle\ \rangle;$$

the number of objects is exponential. The positive effect of this phenomenon is that, only the correct copy of the object is used when it is invoked by the public output function. Just like the SME approach, the execution of multiple branches can have unintended consequences in a dynamic approach. The ZaphodFacets[2] is an implementation of the faceted approach as a plug-in in Firefox. It use the Narcissus JavaScript engine[3]. Cross-site scripting is handled effectively by this approach by assigning each domain into separate principal. Variables from each domain are accessed only if there is access to that principal hence significantly reducing the effects of XSS.

The ASD was proposed by Deepak et. al. [12] and forms the first basis for the approach described in this paper. In this approach each secret variable is split

[2] `https://github.com/taustin/ZaphodFacets`
[3] `http://en.wikipedia.org/wiki/Narcissus_(JavaScript_engine)`

into public and private values. ASD is similar to the faceted approach in terms of having a single process and different values for private variables. However, the key difference is that a variable can have only one private value in ASD while the faceted approach could force it to have multiple private values. Further, ASD does not execute additional branches based on the split values.

ASD seems to provide a more fine-grained IFC with function level control and its performance degradation is much more acceptable than competing approaches [12]. However, this approach does not adhere to TINI; though the secret is never given as a public output. Therefore, it suffers from lower security guarantees than SME while offering being practical. The core of this model starts with a split variable which is represented as $\lceil \text{public}^p \parallel \text{private}^s \rfloor$. The split variable consists of a public value and a private value which are stored in different memory locations. The default symbol table connects the variable to its public value while the ASD mechanism overloads the symbol table at appropriate junctures to change the inferred memory location. This information is maintained in a data-structure called a dictionary that is unique to each function defined in the policy.

ASD's policies allow a differentiation to read and write accesses to secret variables. These policies hence result in more fine-grained IFC which we find to be suitable as a base for our model. The working of ASD is shown in the figure 1. There policies and the JavaScript program are the input. The monitor which is added to the JavaScript compiler interprets these policies and creates data-structures called dictionaries. The variables which contain secret values are split to show public and private parts. The private parts of the variable are inferred from the dictionaries. These dictionaries are updated by the monitor according to information flow.

However, the tracking mechanism used in this IFC has been shown to be less efficient in write operations in comparison with read operations by the authors themselves. Further, the IFC mechanism does not consider all possible information flows and relies solely on over-approximation. We believe that with some suitable changes to this model supplemented by learning, ASD could eventually become adherent to TINI while becoming more efficient.

3 Description of our approach

As described in the section 1, our work is an extensio of the ASD approach.Therefore, we describe the preliminaries regarding ASD in subsection 3.1. This is followed by the description of the dependency graph which tracks the various secrets in our approach in subsection 3.2. We finally discuss the evolution of the dependency graph with information flows in subsection 3.3.

3.1 Preliminaries on ASD

ASD is an IFC where secret variables are split to store two different values in their private and public addresses separately. The access to the secret value for

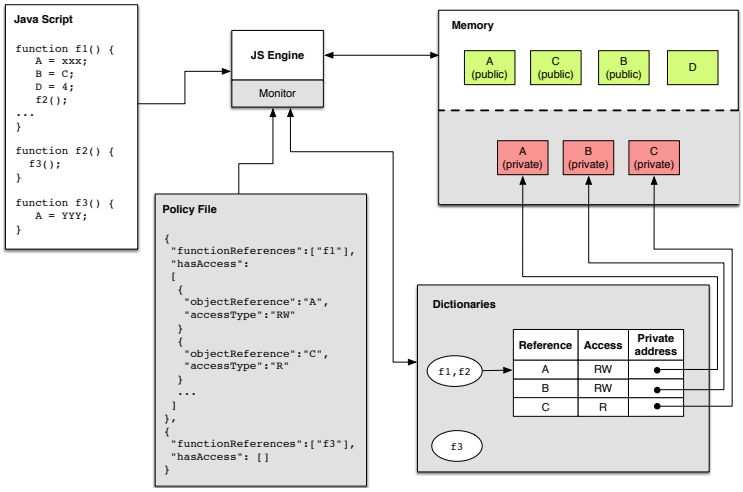

Fig. 1: Address Split Design [12]

a given variable is determined by the function that refers to the variable. The authors propose a dictionary data-structure to track the secret variables based on the information flow. A mechanism called the dependency tracker (`DT`) keeps track of the current statement and the list of secret values that are being used in the current execution. The dictionaries are changed based on the `DT` at every write into a variable.

An example of ASD's working is shown in listing 1.1.

```
1 var a = 2;
2 function f1()
3 {
4   a = 3;
5   print(a);
6   var b = a;
7 };
8 f1();
```

Listing 1.1: ASD Example

Let us consider that variable `a` is secret and function `f1` has access to `a`. In that case, at the beginning of the function call, $a = \lceil 2^p \parallel \text{undefined}^s \rfloor$. This notation signifies that the variable `a` has been split. Here, `a` has a public value that was initialized to 2 and a secret value that is undefined. However, when executing function `f1`, in line 4, the secret value is changed. This is because `f1` has access to the variable `a`. Therefore, $a = \lceil 2^p \parallel 3^s \rfloor$.

In line 6, the variable `a` is read. Therefore, the dependency tracker is updated and `DT = {a}`. When variable `b` is initialized, it is split. The public value remains as `undefined` while the secret value is assigned as `3`. Therefore, $b = \lceil \text{undefined}^p \parallel 3^s \rfloor$.

Further, any function that has access to `a`, is given access to `b`. Hence, the function `f1`'s dictionary would be:

Variable	Rights	Address
a	RW	@(A)
scope(f1).b	RW	@(B)

Table 1: Dictionary of function f1

In the table 1, the notation "scope(f1).b" represents the local variable of the function. Each run of the function generates a unique scope identifier which is used to identify which instance of the local variable is being used. Hence, the use of the scope identifier allows handling of local variables in all cases including recursion.

Any function that is defined in the policy is called a self-sufficient function and has its own dictionary. All other functions are called utility functions and use the dictionary of the self-sufficient function that called them. To implement the above mechanism, the symbol table is overloaded with a dictionary data-structure in ASD. There is hence a unique dictionary for every function defined in the policy.

ASD creates all dictionaries when interpreting the policies and changes every dictionary based on the monitored information flow. This creates an increased overhead when there is a large number of dictionaries. We hence propose another data-structure called the dependency graph to keep track of the information flows supplanting the existing dictionaries. We call this approach Address Split Design with dependency graphs (ASD-DG). When the dependency graph is updated, those changes will be noted by a learning mechanism. These learnt dependencies will be applied to the variables at the end of the function's execution. In the next subsection we will discus the dependency graph and its working.

3.2 Dependency graph

The dependency graph is a tree data-structure which contains three types of vertexes, namely, the function nodes, root nodes, and dependent nodes. The function nodes represent the various self-sufficient functions, whereas the root nodes and dependent nodes represent the various variables that contain secret values. Root nodes are created for all variables represented in the policies. Dependent nodes are created when a variable contains a secret value originating from another variable due to information flow.

In this paper, we describe the formalism along with the concepts involved in our approach. The various initial suppositions are given in `[DECLARATION]`. Here, the various representations used throughout the rest of the paper have been defined. We continue the representation used in ASD as part of the while

language. Most rules defined in ASD hold true to ASD-DG as well. We will describe the rules that change in greater detail in parallel with the dependency graph.

[DECLARATION]

$$
\begin{aligned}
\mathit{Let}, \texttt{Variables} &\ (x \in \mathit{Var}) \\
\texttt{Functions} &\ (f \in F \subset \mathit{Var}) \\
\texttt{Statements} &\ (S \in \mathit{Stm}) \\
\texttt{State} &\ (s(f) \to \{\mathit{public}, \mathit{secret}\}) \\
\texttt{Privilege} &\ (\mathit{Priv} \in \{\mathit{read}, \mathit{write}, \mathit{read} + \mathit{write}\}) \\
\texttt{Policy Specification} &\ (\mathbb{P}(\mathit{Var}, F, \mathit{Priv}) \to \mathit{Boolean}) \\
\texttt{Dictionary} &\ (\mathbb{D} \in \mathit{Dict}) \\
\texttt{Address Space[Var]} &\ (\mathbb{A}[x] : \mathit{Var} \to \mathit{Address}) \\
\texttt{Access Control} &\ \begin{pmatrix} AC_{control}(f,x) : (F, \mathit{Var}) \to \mathit{boolean}\, \mathit{where}, \\ \mathit{control} \in \{\, (r)\mathit{ead}, (w)\mathit{rite}\,\} \end{pmatrix} \\
\texttt{Dependency Tracker} &\ \texttt{dt} \in \texttt{DT} \subset \mathit{Var} \\
\texttt{Dependency Graph} &\ \mathbb{DG} \\
\texttt{Dependency Graph States} &\ \mathbb{DG}\texttt{State}\{\texttt{active}, \texttt{inactive}\} \\
\texttt{Dependency Graph Nodes} &\ \left(\begin{array}{l} \mathbb{DG}_{node}\, \mathit{where}, \\ \quad \mathit{node} \in \left\{ \begin{array}{l} \texttt{root} : \texttt{"Root node"}, \\ \texttt{f} : \texttt{"Function node"} \\ \texttt{d} : \texttt{"Dependent node"} \end{array} \right\} \end{array} \right) \\
\texttt{Reinforced Learning} &\ L, \ \texttt{collection of} \ \{\, (f)\mathit{unction}, (dt), (v)\mathit{ariable}\,\} \\
\texttt{Flow Type} &\ \{FL \in (e)\mathit{xplicit}, (i)\mathit{mplicit}\}
\end{aligned}
$$

The dependency graph is used to maintain data about the various variables that contain secret values due to information flow. It is a tree structure which can be defined as

$$
\mathbb{DG}\texttt{(V,E)}\ \mathit{where}, \left\{ \begin{array}{c} \texttt{V is a set of vertexes} \\ \texttt{V} \ni \{\, \mathbb{DG}_f, \mathbb{DG}_{root}, \mathbb{DG}_d \,\} \\ \\ \texttt{E is a set of edges} \\ \texttt{E} \ni \{\, \mathbb{DG}_f \to \mathbb{DG}_{root}, \mathbb{DG}_{root} \to \mathbb{DG}_d \,\} \end{array} \right\}
$$

The structure of the dependency graph is a simple unidirectional that can only consist of three layers. The edges are strictly only between the function nodes to root nodes or root nodes to dependent nodes. Function nodes can have any number of child root nodes. Similarly, there is no limitation on the number of child dependent nodes for root nodes. Root nodes can have two possible

states, active and inactive. When interpreting the graph, the functions can access variables represented by any of the active root nodes with whom they share an edge. The state modifications occur due to information flow. Such modifications are described in greater detail along with the evolution of the graph in subsection 3.3. Finally, a function can access a dependent node if and only if it has access to all root nodes which are parents to that dependent node. In the formalism, the $x \rightsquigarrow f$ represents that `f` can access x. In the figure 2 `f2` has access to `V6`. However, it cannot access `V5` because `f2` cannot access `V1`.

An example of this data structure, can be seen in the figure 2.

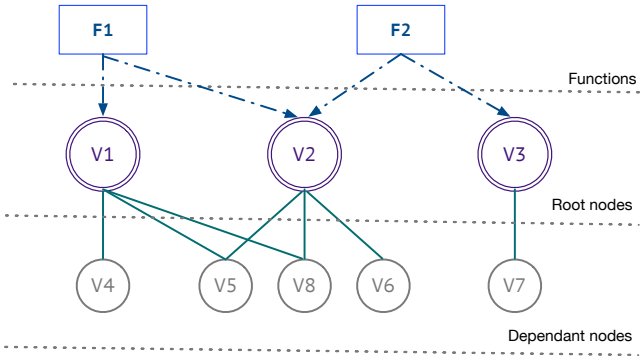

Fig. 2: Dependency graph

When the variables contain secret values because of information flows, they are added to the dependency graph. In our approach, we use the current `DT` to assign parent nodes. The `DT` keeps track of the current set of secret values that are influencing the information flow.

The rule `[POLICY: FUNCTION READ ACCESS]` describes the interpretation of the policy into the dependency graph. The read access constructs the link between the function nodes and the root nodes in the dependency graph. The root nodes are the initial secrets. The other secrets are added to the dependency graph because of information flows from the root nodes and hence become dependent nodes. The main purpose of the dependency graph is to overload the

`[POLICY: FUNCTION READ ACCESS]`

$$\frac{\mathbb{P}(f, x, AC_r)}{\mathbb{DG}_f \leftarrow f; \mathbb{DG}_{root} \leftarrow x; x \rightsquigarrow f}$$

symbol table on a just in time basis as needed by the compiler. When a function

attempts to read a variable, the monitor checks the dependency graph to validate the function's permissions to perform the operation. The rules for accessing the variable are given in [RUNTIME: FUNCTION READ ACCESS]. In this equation, for a given policy, the function and variable are added to the dependency graph and then f becomes a parent of x.

[RUNTIME: FUNCTION READ ACCESS]

$$\frac{f \in \mathbb{DG}_f; x \in \mathbb{DG}\ s.t.\ x \rightsquigarrow f, \mathbb{DG}\texttt{State}(x) = \texttt{active}}{AC_r(f,x) = \texttt{true}; s(f) = secret}$$

$$\frac{f \in \mathbb{DG}_f; x \in \mathbb{DG}\ s.t.\ x \rightsquigarrow f, \mathbb{DG}\texttt{State}(x) \neq \texttt{active}}{AC_r(f,x) = \texttt{false};}$$

$$\frac{f \in \mathbb{DG}_f; x \in \mathbb{DG}\ , \mathbb{DG}\texttt{State}(x) = \texttt{active}, x \not\rightsquigarrow f}{AC_r(f,x) = \texttt{false};}$$

$$\frac{f \notin \mathbb{DG}_f;}{AC_r(f,x) = \texttt{false};} \qquad \frac{x \notin \mathbb{DG};}{AC_r(f,x) = \texttt{false};}$$

3.3 Dependency graph evolution

In this section, we discuss the dependency graph's evolution with the various variables that are added to it over time. Variables evolve when a function performs a write operation. There have been no changes in the write operation from the original ASD in this paper. Since write permissions do not change because of information flow and are only present to protect the variable, they are maintained as a simple list. When a function writes a secret value into a public variable, this variable is split and added as a dependent node to the dependency graph. If this split variable becomes dependent on another root, it is simply moved to become a child of that root node. However, if a root node becomes dependent on another root, a dependent node pointing to that variable is created and original root node becomes inactive. An inactive root node exists only for its children. This implies that the orignal root variable's value has changed but there are other existing variables which contain some information about the value due to information flow.

Let us consider a statement, `V2 = V4 + 4;`. It can be observed in the figure 3 that the variable `V2`, which is a root variable has been changed. The figure shows that by marking the root node `V2` as a grayed out node. This implies that the function does not have access to the variable represented by this node but it may have access to the variables represented by its children. It can also be observed that the variable `V2` is now dependent on `V1` and is indicated by a square in the figure.

In ASD, every dictionary containing the variable required a change when the variable was updated. However, dependency graph needs only a single operation

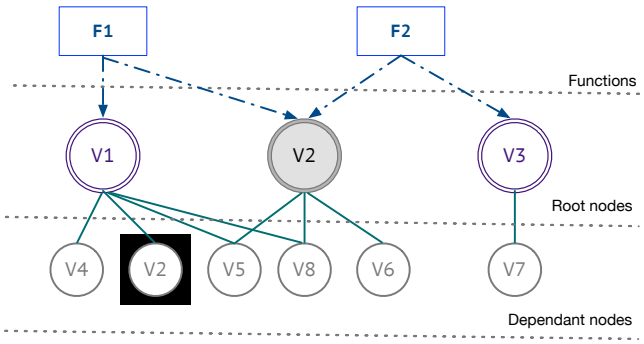

Fig. 3: Dependency graph evolution

to be performed to the same effectiveness. Every time the dependency graph changes, it is registered by the second part of model, the reinforcement learning mechanism. The use and working of the learning mechanism is described in greater detail in the following section.

4 Reinforcement learning

The evolution of the dependency graph is continuously monitored when a function is executed and this data is used to learn about the function's characteristics. The information collected is stored as a persistent data set to be used in subsequent executions. The purpose of this learning is to understand the various paths and loops that were taken in prior executions.

The information collected contains the following: the name of the function, the dependency tracker at the time of the split, the variable being split/updated, root variable and the layout of the relevant nodes in the dependency graph. Once collected, this information is used to split/update variables at subsequent flows.

Let us consider the current state to be as shown as in the figure 2. We now consider functions `f4, f5,` and `f6` as shown in the listing 1.2.

```
// var V1 = secret(true/false);
var V10 = true;
var V11 = true;
function f4()
{
  if (V1)
    V10 = false;
  if (V10)
    V11 = false;
  return V11;
};
function f5(x)
{
  var y = x+1;
  console.log(y);
};
function f6()
{
```

```
19   if(V10)
20   {f5(V10);}
21   else{f5(0);}
22 };
23 f4();
24 f6();
```

Listing 1.2: ASD Example

In this example, the variable `V1` is a secret variable. Here we consider that the functions `f4` and `f6` have read access to `V1`. The function `f5` does not have access to `V1`. If `V1` = true^s, in line 7, the public variable `V10` becomes a secret. In the dependency graph, its root node is `V1`. If `V1` = false^s, in line 9, the public variable `V11` becomes a secret with its root node being `V1` due to the over-approximation of ASD.

```
1 [{function:"f4", rootVariable:"V1", split:"V10",
2                   DT:[V1], dgInfo:[{"V1", "V10"}]},
3 {function:"f4", rootVariable:"V1", split:"V11",
4                   DT:[V1], dgInfo:[{"V1", "V11"}]}]
```

Listing 1.3: ASD Example

The set of variable information shown in listing 1.3 is the persistent data set that is maintained about the function `f4` for the example 1.2. As stated above, it contains information on dependency tracker at the time of the split as well as the root node that was allocated as the parent to the variable at the end of the information flow. Each time a variable is split or its secret value is updated because of a dependency, a data set for the variable is created. This data set is added to the a persistent array of data sets if not already present. This is hence the "learnt data".

Hence, at the end of the execution of function `f4`, it is noted that variable `V10` and `V11` are dependent on the variable `V1`. This information is used for later executions. Now, let us consider the same function such that variable `V1` = $\lceil \text{false}^p \parallel \text{false}^s \rfloor$. In this case, the line 7 is not executed. However, we know from the prior execution that `V10` is dependent on `V1` based on `DT` at the time of the split. Hence, `V10` is split at the end of the execution and its private address contains the value copied from its public address, i.e. `V10` = $\lceil \text{true}^p \parallel \text{true}^s \rfloor$.

Whenever, a variable is split or updated due to information flows, the changes to the dependency graph are noted into the learning mechanism. Therefore, it is represented as part of the rules `[VARIABLE UPGRADE]` and `[VARIABLE UPDATED]`. These rules stipulate the various necessary steps when a new secret value flows into a public variable and split variable respectively. In these rules, the *v.root* represents the parent root node set the variable *v* and *v.root* $\rightsquigarrow x$ implies that the node of the variable x becomes a child node to all the root nodes of variable *v*. The rules show that a `join` operation is performed to the persistent set of learnt data along with a successful variable update or split.

The rationale and the necessity for this split becomes evident in the execution of the function `f6`. The table 2 illustrates the execution of function `f6` if variable `V10` was split or not. It must be noted that `f6` has read access to `V1` according to the policy definition.

[VARIABLE UPDATED]

$$\frac{f \in \mathbb{DG}_f; x \in \mathbb{DG}; x \rightarrow x'; AC_w(f,x) = \texttt{true}; \texttt{dt} = \{\}}{x^s \leftarrow A[x']; x \rightsquigarrow f; L \bowtie (f,x,\texttt{dt})}$$

$$\frac{f \in \mathbb{DG}_f; x \rightarrow x'; AC_w(f,x) = \texttt{true}; \texttt{dt} \neq \{\}}{x^s \leftarrow A[x']; \forall (v \in \texttt{dt}) v.root \rightsquigarrow x; L \bowtie (f,x,\texttt{dt})}$$

$$\frac{f \in \mathbb{DG}_f; x \rightarrow x'; AC_w(f,x) = \texttt{true}; x \in \mathbb{DG}_{root}; \texttt{dt} \neq \{\};}{State(x) = inactive}$$

$$\frac{f \in \mathbb{DG}_f; x \rightarrow x'; AC_w(f,x) = \texttt{true}; \texttt{dt} \neq \{\}; x \in \mathbb{DG}_d}{\texttt{delete } x \in \mathbb{DG}_d}$$

$$\frac{f \in \mathbb{DG}_f; x \rightarrow x'; AC_w(f,x) = \texttt{false};}{x^p \leftarrow A[x'];} \qquad \frac{f \notin \mathbb{DG}_f;}{x^p \leftarrow A[x'];}$$

[VARIABLE UPGRADE]

$$\frac{f \in \mathbb{DG}_f; \texttt{dt} \neq \{\}; S(y); y \notin \mathbb{DG}}{\mathbb{A}[y^s]; y \leftarrow \lceil \mathbb{A}[y] || \mathbb{A}[y^s] \rfloor; (\forall v \in \texttt{dt})\{v.root \rightsquigarrow y\}; L \bowtie (f,y,\texttt{dt});}$$

In the table 2 we show the execution of the function `f6` after the function `f4` has been executed. There are two columns where we compare ASD with ASD-DG post learning has been completed. The two cases where `V1` = true^s and `V1` = false^s have been considered in this example.

The first major difference is caused because of `V10` being split for both values of `V1` in case of ASD-DG post learning. Since the variable split is known, the `DT` remains the same for both cases. However, this is not the case in simple ASD. In the first case, the variable is added to the `DT`. The rule `[SELF-SUFFICIENT CALLED]` [12] is invoked because `f5` is also a self-sufficient function. Since `f5` does not have access to the `V11`, the function call is skipped. This is in line with the dynamic policy enforcement where the secret addresses are not allowed to flow into the public addresses.

[SELF-SUFFICIENT CALLED]

$$\frac{\{S(f); f \in \mathbb{DG}_f; dt = \{x\}; AC_r(f,x) = \texttt{true};\}}{s(f) = secret; dt = \{x\};}$$

$$\frac{\{S(f); f \in \mathbb{DG}_f; dt = \{x\}; AC_r(f,x) = \texttt{false}; FL = i\}}{(Skip\, S(f))}$$

$$\frac{\{S(f); f \in \mathbb{DG}_f; dt = \{x\}; AC_r(f,x) = \texttt{false}; FL = e\}}{s(f) = public; x \rightarrow A[x]; dt = \{\};} \qquad \frac{\{S(f); dt = \{\};\}}{S(f);}$$

function `f6()`	ASD	ASD-DG post-learning
First case [V1=true^S]		

`1. {if(V10 == 1)`
`2. {f5(V10);}`
`3. else{f5(0);}`
`4. }}`
	`true, DT={V10}`
	~~`f5(x);`~~
		`true, DT={V10}`
		~~`f5(x);`~~

Second case [V1=false^S]		

`1. {if(V10 == 1)`
`2. {f5(V10);}`
`3. else{f5(0);}`
`4. }}`
	`false; DT={}`
	
	`DT={}; f5(0);`
		`false; DT={V10}`
		
		`DT={V10};`~~`f5(0);`~~

Table 2: ASD-DG working

The use of the dependency graph is to compensate for the deficiencies caused by the rule `[SELF-SUFFICIENT CALLED]` in ASD. This rule is necessary to prevent the secret from being leaked. However, it still can cause leak on whether the variable was split. This occurs when a self-sufficient function fails to split a global variable because of that branch not being executed and this global variable being used in a subsequent self-sufficient function as a conditional in an implicit flow. In the table 2, `V10` being split affects the `DT` in line 1, therby allowing/denying the execution of `f5`. While such a leak is only caused under specific circumstances, and the actual secret value is never leaked, it nevertheless undesirable and makes the model non-adherent to TINI. We aim to solve this issue through learning. Adding the dependency graph approach, the model gradually closes up its leaks and will eventually become adherant to TINI. Further, since the dependency graph keeps track of the variables for every run, it can also observe dependencies caused by the use of `eval`, every case of a switch case and other information flows gradually over a period of time. Hence, the dependnency graph and how it keeps track of various information flow is very important to the model.

The various information learnt is used by the rule `[DG LEARNING - FUNCTION ENDED]`. At the end of the execution of the function `f`, if there is any past secret information flow, additional IFC actions are performed. The learnt data contains a set of function, dependency tracker and the dependent variable. If the dependent variables have not already been split, a variable upgrade is performed based on the dependency tracker. Else, a variable update is performed.

`[DG LEARNING - FUNCTION ENDED]`

$$\frac{exec(f) \rightarrow \texttt{completed}; L(f) \neq \varnothing}{(\forall r \in L(f))\{\texttt{([VARIABLE UPGRADE]/[VARIABLE UPDATED])(r);}\}}$$

4.1 Eventual TINI

It must be noted that the premise of our model starts with not being adherent to TINI initially. This is because, ASD could leak state information of a variable at the end of the execution of a function. This leak does not mean that secret values would be printed but that it is possible to infer whether a public variable has become a dependent node or not.

`[Proof for TINI]`

$$\texttt{Let,}$$

$$\texttt{Functions}\left(f_h \bigcup f_l \equiv \overset{\forall f \in F}{\bigcup} f\right)$$

$$\texttt{Public output functions}\,(f^p \in f_l)$$

$$\texttt{Secret input}\,x_{in.s}$$

Proof:

$$(\forall f \in f_h) : \{f \leftarrow AC_r(f,x) = \texttt{true};\} \quad (\forall f \in f_l) : \{f \leftarrow AC_r(f,x) = \texttt{false};\}$$

$$(\forall f \in f_h) : \{f \leftarrow AC_w(f,x) = \texttt{true};\} \quad (\forall f \in f_l) : \{f \leftarrow AC_w(f,x) = \texttt{false};\}$$

$$\mathbb{A}[x_{in.s}] := \mathbb{A}[x^p_{in.s}] \quad \mathbb{A}[x_{in.s}] :\neq \mathbb{A}[x^s_{in.s}]$$

$$\frac{\{f(x_{in.s}) \mid f \in f_l;\}}{f \leftarrow V_{ref}(x_{in.s}); \mathbb{DG}(x) \not\leftarrow \mathbb{A}[x_{in.s}]} \quad \frac{\{f(x_{in.s}) \mid f \in f_h;\}}{f \leftarrow V_{ref}(x^s_{in.s}); \mathbb{DG}(x) \leftarrow \mathbb{A}[x^s_{in.s}]}$$

$$\frac{S_{f^p} \leftarrow x_{in.s}; FL = e}{f^p \leftarrow \mathbb{A}[x_{in.s}]; S_{f^p};} \quad \frac{S_{f^p} \leftarrow x_{in.s}; FL = i}{(Skip\, S_{f^p})}$$

$$\frac{\left\{f(x_{in.s}), y \not\in \mathbb{DG}, y \leftarrow x \;\middle|\; f \in f_h;\right\}}{split(y); f \leftarrow V_{ref}(y^s); \mathbb{DG}(y) \leftarrow \mathbb{A}[y^s];}$$

$$\therefore \left\{ S_{f^p}(x^s_{in.s}) \equiv S_{f^p}(({x'}^s_{in.s}) \;\middle|\; x_{in.s} \equiv x'_{in.s} \right\}$$

$$\texttt{However,}\; x_{in.s} \equiv x'_{in.s}\; \texttt{iff}\; \forall x \in s, \texttt{if}\; \{x \in \mathbb{DG}\} \implies \{x' \in \mathbb{DG}\}$$

$$\implies S_{f^p}(x^s_{in.s}) \approxeq s S_{f^p}(({x'}^s_{in.s})\; \texttt{iff}\; \forall x \in s, \texttt{if}\; \{x \in \mathbb{DG}\} \implies \{x' \in \mathbb{DG}\}$$

The `[Proof for TINI]` shows that ASD-DG is only adherent to TINI if the same variables are split at the end of the execution of a given function for different values of the secret. The reinforcement learning model solves this issue by collecting information on the states of the various variables and simulating these states at the end of each function. Hence, over time, all execution paths would be covered and even dynamic flows like eval operations can be handled to a certain extent. Since the learning model would eventually ensure that the states of the variables would eventually be the same, the ASD-DG can be said to be adherent to "eventual TINI".

5 Conclusion

There is a clear and urgent need to address the various information leaks in the context of JavaScript. The dynamic nature of the language makes the standard approaches insufficient or inefficient. In this case, starting with a practical efficient approach and making it adhere to more restrictive security guarantees over time is more appropriate. We have proposed one such model and take the novel approach of using learning in the runtime environment to achieve our goals of "eventual TINI". In this process, we have also proposed a proper change to the architecture of ASD to become more efficient. Taking ASD as the base has also allowes us to account for fine-grained function level policies. We hence, propose this model as a viable IFC model for the modern web-browser. The future work of the model is to work towards a complete web-browser implementation of the proposed model.

References

1. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: Proceedings of the 13th European Symposium on Research in Computer Security. pp. 333 – 348. Springer-Verlag Berlin, Heidelberg (2008)
2. Austin, T.: Dynamic information flow analysis for Javascript in a web browser. Ph.D. thesis, University of California, Santa Cruz (2013)
3. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. ACM SIGPLAN Notices 44(8), 20 (Dec 2009)
4. Bell, D., LaPadula, L.: Secure Computer Systems : Mathematical Foundations. Tech. rep., dtic.mil (1973)
5. Biba, K.J.: Integrity Considerations for Secure Computer Systems. Tech. rep., The Mitre Corporation (1975)
6. Bielova, N.: Survey on JavaScript security policies and their enforcement mechanisms in a web browser. The Journal of Logic and Algebraic Programming 82(8), 243–262 (2013)
7. Devriese, D., Piessens, F.: Noninterference through Secure Multi-execution. 2010 IEEE Symposium on Security and Privacy pp. 109–124 (2010)
8. Groef, W.D., 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. ACM, Raleigh, North Carolina, USA (2012)
9. Hedin, D., Sabelfeld, A.: Information-Flow Security for a Core of JavaScript. In: 2012 IEEE 25th Computer Security Foundations Symposium. pp. 3–18. IEEE (Jun 2012)
10. Kashyap, V., Wiedermann, B., Hardekopf, B.: Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. In: 2011 IEEE Symposium on Security and Privacy. pp. 413–428. IEEE (May 2011)
11. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (Jan 2003)
12. Subramanian, D., Hiet, G., Bidan, C.: Preventive information flow control through a mechanism of split addresses. In: ACM 9th International Conference on Security of Information and Networks 2016. ACM (july 2016)