Practical Information Flow Control for Web Applications

Angel Luis Scull Pupo, Laurent Christophe, Jens Nicolay, Coen de Roover, and Elisa Gonzalez Boix

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
{ascullpu, lachrist, jnicolay, cderoove, egonzale}@vub.be

Abstract. Current browser-level security solutions do not provide a mechanism for information flow control (IFC) policies. As such, they need to be combined with language-based security approaches. Practical implementations for ICF enforcement remains a challenge when the full spectrum of web applications features is taken into account (i.e JavaScript features, web APIs, DOM, portability, performance, etc.). In this work we develop GIFC, a permissive-upgrade-based inlined monitoring mechanism to detect unwanted information flow in web applications. GIFC covers a wide range of JavaScript features that give rise to implicit flows. In contrast to related work, GIFC also handles dynamic code evaluation online, and it features an API function model mechanism that enables information tracking through APIs calls. As a result, GIFC can handle information flows that use DOM nodes as channels of information. We validate GIFC by means of a benchmark suite from literature specifically designed for information flow verification, which we also extend. We compare GIFC qualitatively with respect to closest related work and show that GIFC performs better at detecting unwanted implicit flows.

Keywords: Information Flow Control · JavaScript · Runtime Monitoring · Browser Security · Programming Language

1 Introduction

Large parts of many contemporary client-side web applications are implemented in or compiled to HTML and JavaScript. In these web applications, developers reuse content, code, and services provided by third parties to avoid reimplementing everything from scratch. The default code inclusion mechanism in web applications are `script` elements that point to a resource providing JavaScript source code. The code a browser downloads in this manner is, however, executed in the same environment and with the same privileges as the code provided by the hosted page itself. This implies that, without additional measures, third-party JavaScript code may have access to sensitive data provided by users. For example, consider a web application including a *password strength checker* component to provide users with visual feedback about the quality of their password. In order for the component to perform this task, it must be provided with

the password value. However, nothing prevents the component from leaking the password to other third party code.

To help mitigate exploits of such security vulnerabilities, modern browsers provide mechanisms such as *Same-Origin Policy* (SOP) and *Content Security Policy* (CSP). SOP allows to isolate content from different web origins coexisting within the same web page [1], but it does not apply to the `src` content of `script` tags. On the other hand, CSP enables developers to specify from which domains the browser can load resources [33, 34], but it does not prevent white-listed third-party components that access users data from leaking this data [25,34]. As a result, SOP and CSP must be complemented with application-level security mechanisms to ensure data privacy and confidentiality.

The goal of *Information Flow Control* (IFC) is precisely to enforce data confidentiality and integrity guarantees in software systems. In this paper we focus on dynamic IFC analysis through runtime monitoring for web applications. Dynamic analysis is said to be more suitable for JavaScript than static verification since statically approximating the behaviour of programs is particularly difficult given the dynamic nature of JavaScript [3, 8]. However, several JavaScript language features still make dynamic IFC analysis a particularly challenging task [3], being the most relevant ones, how to reason about DOM and other web APIs, `eval`, prototype inheritance and finally, how to handle *implicit flows*, i.e. flows caused by non-executed branches. In this paper we explore a practical dynamic IFC mechanism that tackles all these relevant features without requiring VM modifications.

1.1 Problem Statement

We surveyed recent and relevant dynamic IFC approaches for JavaScript that have a publicly accessible implementation and are described in related work: IF-transpiler [32], JSFlow [19], ZaphodFacets [6], FlowFox [14,16], and Jest [13]. In general, existing work aims at tackling some of the aforementioned language feature challenges while keeping the performance penalties at a reasonable level.

	`eval`/DOM	Libraries	Permissive	Portable	Performant
IF-transpiler			✓	✓	✓
JSFlow	✓	✓			
ZaphodFacets			✓		
FlowFox	✓	✓	✓		
Jest	✓	✓		✓	✓
Gifc	✓	✓	✓	✓	✓

Table 1. Overview of recent and relevant dynamic IFC approaches for JavaScript.

Table 1 summarizes our survey of dynamic IFC approaches. Only JSFlow, FlowFox, and Jest offer support for the DOM and `eval`, with Jest requiring

a server-side component to handle `eval`. Yet, support for the DOM and `eval` is crucial when analyzing web applications, because the DOM models an important part of the application state and `eval` is widely used in web applications [22, 30]. Moreover, only those three approaches support modeling the behavior of (external) libraries in terms of information flow.

Permissiveness is considered to be an important factor in making IFC practical [18, 32]. In this regard, JSFlow and Jest are not as permissive as IFtranspiler, meaning that these approaches will prematurely end a secure program execution. On the other hand, FlowFox is permissive.

In terms of performance, applying JSFlow, ZaphodFacets, or FlowFox incurs a high performance penalty because JSFlow and ZaphodFacets add a complete layer of interpretation between the application code and the underlying JavaScript runtime, while FlowFox relies on multiple executions of an application. Overall, approaches that modify the VM (FlowFox) or develop a new interpreter (ZaphodFacets and JSFlow) are expensive.

Finally, JSFlow, ZaphodFacets, and FlowFox are not portable, as they are tied to a particular implementation of a JavaScript or browser environment, greatly diminishing their applicability in a landscape of JavaScript and web standards which is constantly evolving.

In this paper we present Gifc, a permissive and portable dynamic IFC mechanism with support for dynamic code evaluation, external libraries and DOM. Gifc exhibits the following properties:

Support for `eval`. Gifc handles dynamic code evaluation online. This is possible because we employ an instrumentation platform running alongside the instrumented program.

Support for libraries. Gifc features an API function model mechanism that enables information tracking through APIs calls. To handle external function calls we took inspiration from the specification of function models described in [21].

Permissive. The monitor of Gifc is based on the *permissive upgrade* (PU) technique of Austin and Flanagan [5].

Portable. Gifc does not require modifications to the underlying JavaScript interpreter or rely on a specific JavaScript runtime environment, but instead works with any ECMAScript 5 compliant JavaScript interpreter.

Performant. The monitor of Gifc is inlined in the source code, so that the instrumented program (including the monitor) can still benefit from the optimizations offered by the underlying JavaScript runtime.

To the best of our knowledge, the combination of these properties are novel and ensures that Gifc is well-suited to perform practical information flow control for contemporary web applications.

The remainder of this paper is structured as follows. Section 2, informally introduces the key IFC concepts. Next, Section 3 introduces the principal aspects of our approach that drove the implementation of Gifc described in Section 4. In Section 5, we evaluate qualitatively Gifc based on a benchmark suite from literature. We also did a quantitative study to evaluate the performance of Gifc

with respect to the state of the art. Finally, Section 6 compares the features introduced in Section 3 with the state of the art on dynamic IFC.

2 Background Information on IFC

IFC can be used to enforce data privacy and integrity guarantees in software systems [20]. The semantic foundation for IFC is based on the concept of *noninterference* [17, 20]. This property holds for an application when, given the same public inputs, the variation of its secret inputs does not affect its public outputs. Dynamic IFC mechanisms track the dissemination of program values as they are produced and combined during program execution to prevent the flow of a sensitive value to a *public sink* [20].

An IFC policy defines *labels* that express the security level of program values, and identifies the sources that produce values with a particular label. For example, a *low* label `L` can be associated with non-sensitive program values that are allowed to be publicly observable. In contrast, *high* labels `H` can be associated with sensitive values that should remain private to the application. Additionally, an IFC policy identifies *information sinks* in a program and associates them with a label as well. IFC then only allows values flowing into a sink that are less sensitive than that sink's label. An IFC policy therefore establishes how the different security levels are related, for example through the use of a total or partial order (lattice) between labels. In our example, we would have `L` $\subset$ `H`, expressing that `H` is more sensitive than `L`, so that `H` values are not allowed to flow to `L` sinks.

Explicit and implicit flows. Information flows can be categorized into two types [15, 20]. *Explicit flows* arise from the direct copy of information. For example, the assignment expression `y = x` causes an explicit flow from variable `x` to `y`, and after the assignment `y` will have the same value with the same label as `x`. *Implicit flows* arise from control flow structures such as `if`, `return` in a non-final position, `break`, `continue`, and `throw`. For example, after executing the statement `if (z) y=0 else y=1` the value of variable `y` depends on the value of `z`. This results in an implicit flow from `z` to `y`, and after the `if` statement the value of `y` will have the same label as `z`.

Permissiveness. Permissiveness can be understood as the ability of a monitoring mechanism to allow the execution of semantically secure programs [13, 18]. Implicit flows from private variables holding secret values to public variables holding non-secret values enables attackers to infer information about these secret values. Austin and Flanagan [4] proposed the *No-Sensitive Upgrade* (NSU) technique, in which any side effect that depends on secret information will terminate the execution. NSU monitors, however, make a coarse approximation of the all paths of executions of the program in order to ensure soundness. For example, consider the program in Listing 1.1. NSU terminates the execution when it reaches the assignment to `y` because the occurrence this side-effect depends

on the secret value of variable x. Although this behavior is sound, the termination of the program execution is premature since the value of y is never used afterwards.

Listing 1.1.

```
let x = true; //H
let y = false;//L
let z = true; //L
if (x) {y=false};//P
print(z)
```

Listing 1.2.

```
let x = true; //H
let y = false;//L
let z = true; //L
if (x) {y=false};//P
print(y)
```

Listing 1.3.

```
let x = false;//H
let y = false;//L
let z = true; //L
if (x) {y=false};
print(y)
```

Permissive Upgrade (PU) [5] is an alternative to NSU that provides a more permissive approach to handle implicit flows. A PU monitor keeps track of secret-dependent values by means of a special label P that indicates that the information is *partially leaked*, i.e., it is currently secret but in other executions may remain public. The execution is terminated only when a partially leaked value is used in a conditional statement or flows to a public sink. Therefore, at the assignment to y in Listing 1.1, instead of stopping the execution as a NSU monitor would, a PU monitor tags the value of variable y with P and execution continues until the end. However, a PU monitor would halt the execution of the program in Listing 1.2 when reaching the `print` statement. For completeness, we mention that both NSU and PU deem the execution of the program in Listing 1.3 to be safe, although there is an implicit flow from variable x to y. Therefore, these monitors are able to enforce *termination-insensitive* noninterference (TINI) which is weaker than *noninterference* [4, 5, 10].

3 GIFC

This work introduces GIFC, a permissive and portable inlined monitoring mechanism that supports the DOM and dynamic code evaluation and offers support for modeling external libraries. To the best of our knowledge, this combination of properties for a dynamic IFC approach is unique. Before delving into how GIFC offers all the properties from Table 1, we first lay out the attacker assumptions.

3.1 Attacker model

We adhere to the *gadget attacker model* [7]. We assume the user visits a trusted web application in a legitimate browser. The attacker is somehow able to run his malicious JavaScript code on the trusted site, for example because the application includes a script from the attacker's server or by using an improper sanitized input. The attacker does not have any network privileges that allows them to mount a *man-in-the-middle* attack [2]. The only outputs the attacker can observe are those sent to his own server. For that, he can use APIs in the browser environment (e.g `XMLHTTPRequest`). Those APIs are considered sinks of information. Therefore, the duty of the IFC monitor is to prevent the flow of any high or sensitive value to those sinks.

3.2 Permissiveness

GIFC's monitor is a flow-sensitive variation on the PU strategy introduced in Austin et al. [5]. GIFC proposes to use AST information of the program to extend the *pc* label context of language constructs such as `return`, `break`, `throw`, etc., when their execution depends on secret values. This information is crucial and must be handled carefully by approaches like `NSU` or `PU` to ensure soundness and permissiveness guarantees. If the aforementioned language features are not handled, the monitor will potentially leak information and hence, will become unsound. On the other hand, if they are used with an approach like `NSU`, the monitor could become excessively restrictive. For example, consider the code snippet in Listing 1.4 in which `h` is secret. The execution of lines from 2 to 4 depends on the value of `h`, given the `throw` statement will execute based on the value `h`. Therefore, a `NSU`-based monitor will stop the execution at the assignment statement (line 2). In this example this problem is extended until the program encounters the first error handler.

Listing 1.4.

```
1 if (!h) {throw new Error()};
2 y = z;
3 f();
4 g();
```

3.3 Portability

GIFC does not rely on a modified VM like [9,16], nor provides an IFC-aware interpreter like [6,19] since those solutions are inherently not portable. Instead, GIFC relies on code instrumentation. Similarly to [12,13,27,31,32], GIFC inlines the monitor within the program.

Inlining the monitor in the program source code has, however, security implications given that the program runs alongside the monitor and an attacker may attempt to tamper with the monitor state to compromise security. To increase the monitor's security, GIFC and JEST obfuscates all variables names introduced by the monitor. This is a naive approach because an attacker can use the reflective capabilities of JavaScript to inspect and modify the monitor state. A possible way to ensure the security of the monitor could be by means of *freeze/seal* of ECMAScript 5. These functions can be used to protect the monitor which will prevent an attacker from altering the monitor functionality. This will imply freezing `Object, Array, String` and other objects from the standard library. Also, all the object in the prototype chain of the monitor should be secured to prevent an attacker from tampering with its prototype chain. Nevertheless, the security of our monitor is ongoing work.

3.4 Eval

Function `eval` allows the execution of arbitrary code represented by a string value. Existing dynamic and hybrid approaches that rely on source code instrumentation do not support `eval()`. For a source code instrumentation approach

to support `eval()` with minimum performance implications, the instrumentation mechanism must run alongside the instrumented program. In GIFC, we specialized `eval()` to track information flow on the string value that this function receive as argument. Since our code instrumenter is part of the execution environment, when `eval()` is called, its argument is instrumented before its evaluation.

3.5 External library calls

JavaScript web applications do not live in isolation in the browser, but they instead interact with the rest of the system in order to do something useful like processing user input/output, sending data over the network, displaying a web form, etc. All these interactions performed by the application are done by means of calls to web APIs, implemented by the browser in other languages (e.g., C++).

Listing 1.5 shows an example of an external function call, `Math.pow`. When executing that code with GIFC to track the flow of information, the application is actually running with augmented semantics, e.g. values are labeled and monitored. Since external libraries do not understand the values used in the augmented semantics, the monitoring mechanism should not pass label information to `Math.pow`. However, after the external library call, the monitor cannot know which label assign to `x`'s value.

Listing 1.5.

```
let y = 13; //H
let x = Math.pow(y,2);
```

A conservative approach to solve this problem is to label the result value with the most sensitive label of the values involved in the call. However, this is considered to be restrictive [19]. To solve this problem in GIFC, we defined an API function model with two functions φ and γ, inspired by the ones presented by Hedin et al. in [21]. The φ knows how to marshal the values from the monitored program to the external function. Also, it has to store the label all values involved in the call. Those stored labels are then used by γ to decide which label should be attached to the return value of the API function call.

3.6 Document Object Model

The Document Object Model (DOM) is the main web API offering page rendering and input/output facilities [24]. DOM elements are exposed to JavaScript as objects. However, their semantics is different from regular ordinary JavaScript objects. Properties of DOM elements are actually pairs of getter/setter functions provided by the browser that cannot handle labeled values.

Monitoring flows from the DOM is crucial as attackers could store secret information as DOM element or as part of their properties or attributes to then later retrieve them and leak that information.

To be able to reason about the DOM without VM modifications, JavaScript proxies [28] seem a good approach to enhance DOM elements operations with

information flow control semantics. However, the DOM is unable to handle proxified nodes because type checks that inspect actual DOM elements will fail for proxies. Also, our function model from Section 3.5 is stateless, while many DOM elements model state.

In order to monitor the DOM API, GIFC associates a meta-object with each DOM element. This meta-object keeps track the element properties' labels and is stored in its target DOM object as an "anonymous" property, using a symbol property key. Note, however, that this approach is transparent but not tamper-proof. This is because the attacker can gain access to the meta-object by mean the language reflective features (i.e `Object.getOwnPropertySymbols()`).

3.7 Performance

As explained in the introduction dynamic IFC incurs on non-neglegible performance penalties. In particular, the performance of FLOWFOX depends on the number of security level and the number of cores of the CPU given that the program needs to execute once per each security level. On the other hand, providing an IFC aware interpreter like JSFLOW and ZAPHODFACETS incurs in a big performance penalty (as we also later show in Section 5.2).

Similar to IF-TRANSPILER, GIFC employs code instrumentation to inline its monitor within the target program. Inlining the source code is potentially better performant than the aforementioned solutions since the resulting code can benefit from JIT compilation as pointed out in [13]. Section 5 evaluates this research statement and measures the impact of GIFC on the original application.

4 Implementation

We implemented GIFC [1] as a JavaScript framework that takes a JavaScript or an HTML page as input program. GIFC then inlines the IFC monitor by instrumenting the source code of the application. More precisely, the JavaScript code is instrumented to trap relevant operations and call the monitor through a well-defined interface, decoupling the monitor from the instrumentation platform used. Our current prototype assumes that developers tag the sources and sinks in the input program and provide the specification of function models to handle external libraries. In what follows, we will first briefly introduce the used code instrumentation platform, and then provide details on the monitor, how it deals with implicit flows and non-JavaScript APIs.

4.1 Code instrumentation platform

GIFC uses *Linvail* [11] as code instrumentation platform for implementing its monitor. More precisely, it employs Linvail's source-to-source transpiler for JavaScript called *Aran* [2]. Aran takes as input a target program and an analysis

[1] https://gitlab.soft.vub.ac.be/ascullpu/guardia-ifc
[2] https://github.com/lachrist/aran

and produces an instrumented JavaScript program that can be executed on any ES5-compliant interpreter. The analysis is a JavaScript file that describes how JavaScript operations should be embellished. In the case of GIFC, the analysis file provides the traps for language operations (function calls, variable assignment, object property access, etc.) that require calling the IFC monitor.

4.2 Monitor

The instrumented code interacts with the monitor using a well-defined interface shown in Figure 1, distilled from the semantics of the PU monitor presented by Austin et al. [5]. The monitor interface decouples its implementation from

	Monitor function	Description
callbacks	`pushContext(x, t)`	Push a context label given a type
	`popContext(t)`	Pop a context label given its type
	`join(x,y)`	Returns the least upper bound of the labels
	`permissiveCheck()`	Determine if there is no PU violation in a branching point
	`enforce(y,...xs)`	Enforce IFC if `y` is a sink and some of `xs` is a source
	`leave(fn)`	Remember all values' labels of an external function call before its execution
	`enter(fn, val)`	Attach a computed label to the return value of an external function
impl	`tagAsSource(x)`	Tags `x` as source (i.e. sensitive data)
	`tagAsSink(x)`	Tags `x` as sink (i.e. produce a public observable data)
	`addFnModel(`φ`, `γ`))`	Registers a model γ for an external function φ

Fig. 1. Monitor interface

the instrumentation platform, which enables changing parts of the monitoring mechanism independently. We would also like to exploit this decoupling in future work to experiment with other code instrumentation platforms.

Figure 1 distinguishes two categories of monitor functions. Calls to the functions marked as "callbacks" are automatically inserted into the target program during code instrumentation (see Section 4.4). Calls to the functions marked as "impl" have to be manually called by the IFC implementor, i.e. developer performing IFC analysis. Those calls refer to the tagging functions for tagging sources and sinks, and to add a function model (see Section 4.3).

4.3 Implementer monitor functions

GIFC provides functions `tagAsSource` and `tagAsSink` that developers have to insert into a program to identify sensitive sources and sinks. For example, the program in Listing 1.6 shows the required tagging for enabling the IFC monitor to prevent the flow of the user password to the browser console output. Function `console.log` is tagged as a sink, and the `value` property of the HTML element with id `#pass` as a source.

Listing 1.6. Prevent password leakage

```
tagAsSink(console.log);
const onClickHandler = () => {
        const $ = document.querySelector;
        let pass = tagAsSource($('#pass').value);
        ...
        console.log(pass);
}
```

Although developers currently have to manually tag sources and sinks in the code, it would be possible to devise a more declarative (external) manner for specifying sources and sinks, which the code instrumenter can then use to introduce the tag functions in the target program where appropriate. We are currently building plugin support enhanced with AI machinery to automate the marking sources and sinks in the future.

Besides identifying sources and sinks, GIFC also expects that external functions are registered using `addFnModel(fun, γ)`. Function γ has to approximate the flow of information of function `fun` in terms of the labels of the arguments. For example, for `Math.pow(x,y)` shown in Listing 1.5, we would register `γ(x,y) = x⊔y`, correctly capturing the notion that if `Math.pow` is called with one or two sensitive argument values, then the resulting value is also sensitive. We implemented models for some objects of the standard libraries including `Math`, `Array`, and `String`. However, the monitor fallback to default conservative model for functions calls that do not have precise model implementation.

4.4 Callback monitor functions

GIFC uses a shadow stack to maintain the *pc* label. The `pushContext()` function pushes a security label into the stack every time the program encounters a control flow statement. The label value is the `join` of all values that influences control flow in a control flow statement.

`popContext()` removes the top element of the stack when the execution reaches the end of a control flow structure body.

Our monitor actually maintains an *exception* stack that keeps track of implicit flows that arise from throwing exceptions in sensitive contexts. We push into the *exception* stack when the execution of a `throw` statement depends on sensitive information. This is because there is no syntactic way to know when an exception will be handled. Then, when a `catch` handler is reached, we pop all values from the *exception* stack.

The `join(a,b)` operation is used whenever the label of a value depends on multiple values (i.e. the *least upper bound* of the elements). As a concrete example, consider `let z = x + y;`. The label of `z` depends on the more sensitive label involved in the values of the binary operation (also, in the label of the pc context, etc.).

The `permissiveCheck()` enforces the PU invariant at the branching point of control flow structures to avoid total leak of information. `enforce()` is then

used at code locations (e.g function application, setters) where information can leak the system to prevent information flow violations. It checks if there is any sensitive value flowing to a setter or function annotated as sink.

Functions `addFnModel(fun, γ)`, `leave`, and `enter` enable the IFC monitor to interact with non-instrumented functions, i.e. external function calls. As mentioned, external functions need to be registered using `addFnModel(fun, γ)`. During program execution, upon the call to an external function, function `leave` looks up the corresponding γ function, splits the labels from the argument values and applies γ, and stores the resulting label ℓ. Next, the actual external function is called with the unlabeled argument values. Finally, function `enter` attaches the stored label ℓ to the value returned from the non-instrumented function call.

Recall that to reason about the DOM, GIFC associates a meta-object with each DOM element. When a getter or setter is executed on a DOM element, the instrumentation ensures that each element property write operation updates its corresponding label in the meta-object, while every value resulting from a property read operation will be labeled with its corresponding label. For handling DOM elements methods, the function model associated to the method is used.

5 Evaluation

In order to evaluate our approach, we performed a qualitative and quantitative evaluation of our GIFC implementation. The qualitative evaluation provides an indication of how effective our approach is in detecting illicit information flows. The quantitative evaluation shows the performance implications of our approach to an uninstrumented baseline and compares it to related approaches.

5.1 Qualitative evaluation

To evaluate the effectiveness of GIFC in a practical way, we compare it with IF-TRANSPILER, JSFLOW, and ZAPHODFACETS by determining whether or not illicit flows are detected in a suite of benchmark programs[3]. The benchmark suite was designed by Sayed et al. [32] and consists of 33 programs specifically designed for testing information flow control. It contains a wide variety of (combinations of) language features that challenge any IFC approach. We extended the benchmark with 5 new programs to test features such as `eval`, API function calls, and property getters/setters not present in the original one. In the GIFC repository we describe the 28 programs included in the original benchmark suite[4]. The last entries in the table describe the new 5 additions. Each benchmark program takes as input a secret string value, which the program attempts to leak explicitly or implicitly in various ways. We ran all tools on all benchmark programs in

[3] Unfortunately we were unable to set up a functional test environment for FLOWFOX and JEST. In the case of JEST certain models are required that are undocumented and not trivial to develop.

[4] https://gitlab.soft.vub.ac.be/ascullpu/guardia-ifc

NodeJS, except for ZaphodFacets, of which the experiments were performed in *Mozilla Firefox 8.0* as required by the tool.

Table 2 shows how Gifc compares to the other three IFC approaches. The ✓ means that a tool was able to detect the illicit information flow, while ✗ indicates that a tool was unable to detect the illicit flow. *R.Err* indicates that a tool threw a runtime exception and was unable to execute the program properly. *In.Err* indicates that the tool was unable to inline the monitor into the original program source code. *Exp* indicates that the tool threw an exception at a point where an illicit information flow could be. However, in these cases it was premature because at that point there was no invalid information flow. This observation was also made in [32].

The results in Table 2 show that Gifc is able to detect and prevent illicit information flows in all test programs. For the 28 programs from the original suite we were able to reproduce the findings reported by Sayed et al. [32] for IF-transpiler, JSFlow, and ZaphodFacets. For the 5 test programs that we extended the suite, Gifc and JSFlow successfully detected all illicit flows. Both IF-transpiler and ZaphodFacets were able to successfully detect an illicit flow in only one out of 5 new test programs.

Adding online support for `eval()` in IF-transpiler needs the static analysis component and the transpiler in the same process of the application. Supporting APIs will require the refactoring of the transformation rules to include function models. Also, it will require implementing the mechanism that allows assigning models to APIs functions which need to be configured at runtime.

From this we conclude that Gifc is on par with the existing tools in terms of detecting illicit information flows in the presence of different JavaScript features.

5.2 Quantitative evaluation

We conducted performance benchmarks to measure the impact of Gifc on the performance of the original application (the baseline), and to gauge how our approach compares with IF-transpiler, JSFlow, and ZaphodFacets in this regard. The set of benchmark programs consists of 9 different algorithms used in Sayed et al. [32]. Table 3 shows the time in milliseconds to run the algorithms. More concretely, it reports the average time of 10 executions of each algorithm. Both JSFlow and ZaphodFacets failed to execute the AES algorithm. This was also reported in [32].

The results in Table 3 show that the approaches that rely on code instrumentation (Gifc and IF-transpiler) have a performance impact which is one or more orders of magnitude smaller than the performance impact of approaches that rely on an additional interpreter (JSFlow and ZaphodFacets). IF-transpiler performs better than Gifc, although performance is still comparable. Important sources of performance overhead in Gifc's dynamic monitor are the wrapping and unwrapping of values before and after API calls, and the emulation of implicit calls to functions `toString()` and `valueOf()` due to implicit value coercion in the target program.

Program	JSFlow	ZaphodFacets	IF-transpiler	Gifc
Test 1	✓	✓	✓	✓
Test 2	✓	✓	✓	✓
Test 3	✓	✓	✓	✓
Test 4	✓	✓	✓	✓
Test 5	✓	R.Err	✓	✓
Test 6	Exp	R.Err	✓	✓
Test 7	Exp	R.Err	✓	✓
Test 8	Exp	R.Err	✓	✓
Test 9	Exp	R.Err	✓	✓
Test 11	Exp	R.Err	✓	✓
Test 11	Exp	R.Err	✓	✓
Test 12	Exp	R.Err	✓	✓
Test 13	✗	R.Err	✓	✓
Test 14	✓	R.Err	✓	✓
Test 15	✓	R.Err	✓	✓
Test 16	✓	R.Err	✓	✓
Test 17	✓	R.Err	✓	✓
Test 18	✓	R.Err	✓	✓
Test 19	✓	R.Err	✓	✓
Test 20	✗	R.Err	✓	✓
Test 21	Exp	R.Err	✓	✓
Test 22	✓	R.Err	✓	✓
Test 23	✓	R.Err	✓	✓
Test 24	✓	R.Err	✓	✓
Test 25	✗	R.Err	✓	✓
Test 26	✗	R.Err	✓	✓
Test 27	✗	R.Err	✓	✓
Test 28	✗	R.Err	✓	✓
Test 29	✓	✗	✗	✓
Test 30	✓	R.Err	In.Err	✓
Test 31	✓	R.Err	✗	✓
Test 32	✓	✗	✓	✓
Test 33	✓	✓	✗	✓

Table 2. Effectiveness comparison

Approach	FFT	LZW	KS	FT	HN	24	MD5	SHA	AES
Baseline	4ms	4ms	22ms	3ms	16ms	13ms	2ms	2ms	9ms
IF-TRANSPILER	14ms	11ms	363ms	10ms	327ms	126ms	33ms	29ms	284ms
GIFC	23ms	34ms	747ms	35ms	1238ms	1233ms	31ms	35ms	780ms
JSFLOW	404ms	421ms	5206ms	661ms	5165ms	4371ms	491ms	566ms	fails
ZAPHODFACETS	100ms	188ms	15563ms	145ms	12657ms	6403ms	124ms	197ms	fails

Table 3. Performance benchmarks

6 Related work

In this section, we discuss the most recent and relevant dynamic IFC approaches for JavaScript previously mentioned (IF-TRANSPILER, JSFLOW, ZAPHODFACETS, FLOWFOX) and some additional related work. All but IF-TRANSPILER and JEST are also part of the most recent survey on IFC by Bielova et al. [10].

Sayed et al. [32] introduce IF-TRANSPILER, an hybrid flow-sensitive monitor inlining framework for JavaScript applications. The static component is used to improve the permissiveness of the monitor by collecting at branching points, the side effects and function calls of branches not taken. At a branching point, the static analysis collects all variables that could have been assigned or functions that could have been called in the untaken branch. In contrast to GIFC, IF-TRANSPILER does not offer support for external libraries neither `eval()` nor DOM, which prevent it from being used in a practical scenario. Also, its static analysis do not handle side effects inside the body of function calls in the untaken branches. Therefore, the soundness of the static analysis is compromised.

JSFLOW [19] is an IFC-aware interpreter for JavaScript that uses NSU to handle implicit flows. To relax NSU, JSFLOW uses upgrade instructions for public labels before entering to a more sensitive context. However, this requires programmer intervention to specify where and what the interpreter should upgrade, which can lead to misconfigurations. JSFLOW is not portable, because it needs to be adapted for each JavaScript engine. Also, it has a considerable performance impact due to the addition of a complete layer of interpretation.

ZAPHODFACETS [6] is an IFC-aware interpreter featuring *faceted values* which capture the multidimensional view of a value with respect to confidentiality levels. They provide formal proofs with respect to TINI and also evaluated their as a plugin implementation for the Firefox browser. However, they lack support for DOM and external libraries. They do not support eval and the application performance is heavily affected due to the added interpretation layer. Also, the ZAPHODFACETS portability is limited to the Firefox browser.

Secure Multi-Execution (SME) [16,29] takes a different approach that traditional monitoring approaches for IFC. Programs under SME are executed multiple times, once for each security level, using special rules for input and output operations. The FLOWFOX implementation require large browser modifications in order to synchronize all the executions. Executions that are not allowed to access sensitive information are provided with dummy values representing more sensitive values. Therefore, any leak of information will not release the secrets of the application. However, it is unclear how dummy values can ensure the transparency of the system.

JEST [13] is an IFC monitor inliner for JavaScript implementing NSU like JSFLOW. It uses the concept of *boxes* to associate label information with program values. To allow the program work on boxes, they rewrite the program using special functions for all JavaScript operations (e.g function calls, assignments, etc.). Like GIFC, JEST has a shadow stack to handle unstructured implicit flows. However, JEST implements the NSU technique which requires the intervention of the programmer to indicate the upgrading points. They also rely on an exter-

nal process to handle dynamic code evaluation, which degrades the application performance on calls to `eval()`.

Santos and Rezk [31] were the first that developed an IFC inlining compiler for a core of JavaScript. They proved that the compiler is able to enforce TINI and developed a practical implementation of it. However, their implementation does not cover external libraries neither DOM.

Bichhawat et al. [9] implemented a dynamic IFC mechanism for the JavaScript bytecode produced by Safari's WebKit Engine. They formalize the Webkit's bytecode syntax and semantics, their instrumentation mechanism and prove non-interference. To improve permissiveness, they implement a variant of PU but their work does not support the DOM or other Web APIs.

Le Guernic et al. [23] developed a sound hybrid monitor that enforces non-interference for a sequential language with loops and outputs. The monitoring mechanism is composed by a variation of the *edit automata* [26] and the semantics of monitored executions. It enforces non-interference by authorizing, editing or forbidding the an specific action during the execution.

Magazinius et al. [27] formalized a framework to inline a monitor on the fly for an small language with dynamic code evaluation.

7 Conclusion

We introduced GIFC, a practical portable dynamic IFC monitoring mechanism. GIFC implements the PU strategy to improve the permissiveness of the monitoring. It offers support for DOM and external libraries enabling a practical use of IFC. Having static information at runtime makes it possible to develop a more precise model of implicit flows. GIFC is the first inlining mechanism that supports dynamic code evaluation online.

Benchmarks results show that the performance impact is better than approaches which rely on a IFC-aware interpreter but it is non-neglegible. Nevertheless, we believe that the approach can be used in settings where security plays a key role. Also, GIFC can aid developers if it is used as IFC testing tool at development time.

In spite of the achievements presented here, there are still some challenges that this kind of approaches need to overcome. First, the performance impact needs to be addressed. Second, the monitor state must be secured given the fact that its state is visible to the application.

References

1. Same Origin Policy - Web Security, https://www.w3.org/Security/wiki/SameOriginPolicy
2. Man-in-the-middle attack - OWASP (Aug 2015), https://www.owasp.org/index.php/Man-in-the-middle_attack
3. Andreasen, E., Gong, L., Møller, A., Pradel, M., Selakovic, M., Sen, K., Staicu, C.A.: A Survey of Dynamic Analysis and Test Generation for JavaScript. ACM Computing Surveys **50**(5), 1–36 (Nov 2017)

4. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. PLAS p. 113 (2009)
5. Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. PLAS pp. 1–12 (2010)
6. Austin, T.H., Flanagan, C.: Multiple facets for dynamic information flow. POPL p. 165 (2012)
7. Barth, A., Jackson, C., Mitchell, J.C.: Securing frame communication in browsers. Commun. ACM **52**(6), 83–91 (Jun 2009)
8. Bichhawat, A., Rajani, V., 0001, D.G., 0001, C.H.: Generalizing Permissive-Upgrade in Dynamic Information Flow Analysis. CoRR **cs.CR**, 15–24 (2015)
9. Bichhawat, A., Rajani, V., Garg, D., Hammer, C.: Information Flow Control in WebKit's JavaScript Bytecode. In: Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, pp. 159–178. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
10. Bielova, N., Rezk, T.: A Taxonomy of Information Flow Monitors. POST **9635**(1), 46–67 (2016)
11. Christophe, L., Boix, E.G., De Meuter, W., De Roover, C.: Linvail - A General-Purpose Platform for Shadow Execution of JavaScript. SANER pp. 260–270 (2016)
12. Chudnov, A., Naumann, D.A.: Information Flow Monitor Inlining. In: 2010 IEEE 23rd Computer Security Foundations Symposium (CSF). pp. 200–214. IEEE (2010)
13. Chudnov, A., Naumann, D.A.: Inlined Information Flow Monitoring for JavaScript. In: the 22nd ACM SIGSAC Conference. pp. 629–643. ACM Press, New York, New York, USA (2015)
14. 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 (CCS 2012). pp. 748–759. ACM (2012). https://doi.org/https://doi.org/10.1145/2382196.2382275, https://lirias.kuleuven.be/handle/123456789/354589
15. Denning, D.E., Denning, P.J.: Certification of Programs for Secure Information Flow. Commun. ACM **20**(7), 504–513 (1977)
16. Devriese, D., Piessens, F.: Noninterference through Secure Multi-execution. In: 2010 IEEE Symposium on Security and Privacy (SP). pp. 109–124. IEEE (2010)
17. Goguen, J.A., Meseguer, J.: Security Policies and Security Models. IEEE Symposium on Security and Privacy pp. 11–11 (1982)
18. Hedin, D., Bello, L., Sabelfeld, A.: Value-Sensitive Hybrid Information Flow Control for a JavaScript-Like Language. In: 2015 IEEE 28th Computer Security Foundations Symposium (CSF). pp. 351–365. IEEE (2015)
19. 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. ACM, New York, NY, USA (2014)
20. Hedin, D., Sabelfeld, A.: A Perspective on Information-Flow Control. Software Safety and Security (2012)
21. Hedin, D., Sjosten, A., Piessens, F., Sabelfeld, A.: A Principled Approach to Tracking Information Flow in the Presence of Libraries. In: Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, pp. 49–70. Springer Berlin Heidelberg, Berlin, Heidelberg (Mar 2017)
22. Jensen, S.H., Jonsson, P.A., Møller, A.: Remedying the eval that men do. In: the 2012 International Symposium. pp. 34–44. ACM Press, New York, New York, USA (2012)

23. Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.A.: Automata-Based Confidentiality Monitoring. In: Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, pp. 75–89. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
24. Le Hgaret, P.: W3c Document Object Model (Jan 2005), https://www.w3.org/DOM/
25. Lekies, S., Kotowicz, K., Groß, S., Nava, E.A.V., Johns, M.: Code-Reuse Attacks for the Web - Breaking Cross-Site Scripting Mitigations via Script Gadgets. CCS pp. 1709–1723 (2017)
26. Ligatti, J., Bauer, L., Walker, D.: Edit automata - enforcement mechanisms for run-time security policies. Int. J. Inf. Sec. (2005)
27. Magazinius, J., Russo, A., Sabelfeld, A.: On-the-fly inlining of dynamic security monitors. Computers & Security **31**(7), 827–843 (Oct 2012)
28. MDN: Proxy (Mar 2018), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy
29. Rafnsson, W., Sabelfeld, A.: Secure multi-execution - Fine-grained, declassification-aware, and transparent. Journal of Computer Security **24**(1), 39–90 (2016)
30. Richards, G., 0001, C.H., Burg, B., Vitek, J.: The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications. ECOOP (2011)
31. Santos, J.F., Rezk, T.: An Information Flow Monitor-Inlining Compiler for Securing a Core of JavaScript. In: ICT Systems Security and Privacy Protection, pp. 278–292. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
32. Sayed, B., Traoré, I., Abdelhalim, A.: If-transpiler: Inlining of hybrid flow-sensitive security monitor for JavaScript. Computers & Security **75**, 92–117 (Jun 2018)
33. Stamm, S., Sterne, B., Markham, G.: Reining in the web with content security policy. In: the 19th international conference. pp. 921–930. ACM Press, New York, New York, USA (2010)
34. Weichselbaum, L., Spagnuolo, M., Lekies, S., Janc, A.: Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In: ACM Conference on Computer and Communications Security. pp. 1376–1387. ACM Press, New York, New York, USA (2016)