GUARDIA: specification and enforcement of JavaScript security policies without VM modifications*

Angel Luis Scull Pupo
Vrije Universiteit Brussel
Brussels, Belgium
angel.luis.scull.pupo@vub.be

Jens Nicolay
Vrije Universiteit Brussel
Brussels, Belgium
jens.nicolay@vub.be

Elisa Gonzalez Boix
Vrije Universiteit Brussel
Brussels, Belgium
elisa.gonzalez.boix@vub.be

ABSTRACT

The complex architecture of browser technologies and dynamic characteristics of JavaScript make it difficult to ensure security in client-side web applications. Browser-level security policies alone are not sufficient because it is difficult to apply them correctly and they can be bypassed. As a result, they need to be completed by application-level security policies.

In this paper, we survey existing solutions for specifying and enforcing application-level security policies for client-side web applications, and distill a number of desirable features. Based on these features we developed Guardia, a framework for declaratively specifying and dynamically enforcing application-level security policies for JavaScript web applications without requiring VM modifications. We describe Guardia enforcement mechanism by means of JavaScript reflection with respect to three important security properties (transparency, tamper-proofness, and completeness). We also use Guardia to specify and deploy 12 access control policies discussed in related work in three experimental applications that are representative of real-world applications. Our experiments indicate that Guardia is correct, transparent, and tamper-proof, while only incurring a reasonable runtime overhead.

KEYWORDS

Language design; DSL; Security Policy; Web Security; JavaScript; Reflection; Runtime Enforcement

ACM Reference Format:
Angel Luis Scull Pupo, Jens Nicolay, and Elisa Gonzalez Boix. 2018. GUARDIA: specification and enforcement of JavaScript security policies without VM modifications. In *15th International Conference on Managed Languages & Runtimes (ManLang'18), September 12–14, 2018, Linz, Austria.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3237009.3237025

1 INTRODUCTION

Today, web applications are no longer monolithic, built using in-house code only. Instead, they can be considered as mashups of content and code included from different third-party sites. However, the inclusion mechanism of browsers is all or nothing: all JavaScript code included from different sources has the same privileges to access sensitive resources such as cookies, location, etc. Developers are thus forced to trust any code that they include into a page. This exposes web applications to various security threats of which Cross Site Scripting, Cross Site Request Forgery, and Sensitive Data Exposure are among the most well-known [11, 14, 18].

Efforts have been undertaken at the browser level to mitigate (some of) these security threats by means of *security policies*. A Browser's Content Security Policy (CSP) enables developers to inform the browser about the sources from which the application is allowed to load resources. A Same-Origin Policy (SOP), on the other hand, restricts the content a web page can access to only resources of the same origin. Nevertheless, the implementations of SOP and CSP present inconsistencies across different browsers and can be bypassed [4, 22, 27]. As a result, browser-level security efforts must be complemented with *application-level* security policies to secure web applications.

In this paper, we present an internal DSL called Guardia for specifying and enforcing application-level access control security policies in JavaScript. Guardia combines a *declarative* policy specification language with a *decoupled* enforcement mechanism, making it possible to experiment with different enforcement techniques that do not require VM modifications. To the best of our knowledge this combination is unique in the context of JavaScript web applications. Guardia's default policy enforcement mechanism for access control policies is based on ECMAScript's reflection.

The contributions of this paper are threefold:

(1) introduction of an internal DSL for the declarative specification of security policies in JavaScript;
(2) identification of the possibilities and limits of policy enforcement based only on reflection with respect to security properties such as completeness, transparency, and tamper-proofness;
(3) evaluation of the applicability and performance impact of dynamic reflection-based enforcement on 3 open source web applications and 10 private real-world web applications.

The remainder of the paper is organized as follows. We first survey existing solutions for specifying and enforcing application-level security policies for client-side web applications, and distill a number of desirable features. Section 3 introduces Guardia's specification language and Section 4.1 describes Guardia's modular enforcement API. The main ideas of an enforcement mechanism based on JavaScript's reflective capabilities are presented in Section 4.2. We evaluate the combination of Guardia's specification language and its dynamic enforcement in Section 5.

*Produces the permission block, and copyright information

ManLang'18, September 12–14, 2018, Linz, Austria

ACM ISBN 978-1-4503-6424-9/18/09…$15.00
https://doi.org/10.1145/3237009.3237025

2 RELATED WORK

A security policy restricts application behavior to prevent vulnerabilities from occurring or being exploited. An application-level security policy expresses a program property that must hold during the entire application's execution. Schneider et al. [23] classify security policies in three classes:

(1) *access control policies* restrict what operations principals can perform on objects,
(2) *information flow policies* restrict what principals can infer about objects from observing system behavior, and
(3) *availability policies* restrict principals from denying others the use of a resource.

In this paper, we focus on access control policies[1]. We surveyed existing solutions for specifying and enforcing access control security policies for web applications, including HV[9], CoreScript [13, 29], BrowserShield [20], WebJail [25], ConScript [17], ObjectViews [16], JSand [1], Phung et al. [19], Richards et al. [21] and Drossopoulou et al. [6]. From this survey, we identified three design choices and associated benefits and shortcomings.

2.1 General-purpose vs. domain-specific specification languages

Some approaches express access control policies in a full-fledged *general-purpose programming language* (GPL) like JavaScript or C++[1, 9, 17, 19, 20, 29]. This provides developers with the freedom of using the complete set of features of the host language. However, relying on a GPL for a domain specific concern (security) may introduce more accidental complexity [8].

Designing a *domain-specific language* (DSL) for expressing security policies aims to free policy designers from the accidental complexity of a GPL. Some approaches propose a standalone (external) DSL language for the purpose of expressing security policies, different from the host language of the application (e.g., [6, 13]). Relying on a new language potentially results in more freedom of expressiveness, but at the cost of having to learn the language first.

An internal DSL combines the best of the two worlds, as it provides the flexibility of an external DSL while both the application and its security policy specifications are written in the same host language. This is the approach taken by WebJail for JavaScript and C++, and by ObjectViews for JavaScript.

2.2 Imperative vs. declarative specifications

Access control security policies are usually specified at the granularity of methods and properties of built-in objects. Many approaches propose *imperative* specifications of policies [1, 9, 17, 19–21]. This offers flexibility, but can lead to security misconfigurations and inconsistencies that can be exploited by attacks. The main disadvantage of an imperative specification is that developers are responsible for ensuring that policies are *tamper-proof* (i.e., they cannot be bypassed) and do not contain bugs or errors that result in new vulnerabilities. Additionally, imperative policies are generally difficult to combine and reuse due to the fact that they can assert various overlapping and conflicting concerns [10, 12]. Alternatively, security policies can be *declaratively* specified [6, 13, 29]. A declarative approach offers a well-defined interface for specifying policies, constraining developers to particular patterns for defining a policy. This leads to less error-prone code and frees developers from manually writing enforcement code [12]. However, a declarative policy specification language usually requires policy developers to use new notations for expressing their policies, and additional support for enforcing them in an engine, parser, or compiler. For example, in CoreScript developers describe policies in XML.

ConScript, ObjectViews, and Phung et al. [19] employ an hybrid approach in which policies are specified in an aspect-oriented manner, but security checks are written in an imperative manner. None of these approaches provide a mechanism to combine policies.

2.3 Modified vs. unmodified runtime for enforcement

An enforcement mechanism can be implemented as part of the target runtime by relying on *VM modifications* [9, 17, 21, 25]. A disadvantage of requiring VM modifications is the limited portability of the resulting security mechanism, which must be reimplemented and customized for each target runtime. Because JavaScript is the lingua franca for programming web applications, VM modifications are not a viable option in this context due to the many browser and backend implementations.

Alternatively, enforcement can be achieved by modifying the application by means of meta-programming. Many approaches provide policy enforcement on the fly by employing the host language's runtime *reflective* capabilities [1, 16]. It is possible, however, that the reflective capabilities of a language are too limited to monitor all security-relevant operations. This, in turn, may restrict the types of policies that can be enforced. For example, in JavaScript security policies can only be applied at object level when using proxies, as proxies cannot be used to track primitive values and their operations.

A second option without requiring a modified runtime is to employ *code instrumentation* to rewrite the target program and selectively inject code to protect those points where security is needed. This technique is employed by CoreScript [29] and Virtual Values [3]. However, code instrumentation has a negative impact on performance.

2.4 Coupled vs. decoupled enforcement

In many imperative approaches developers *mix* the code specifying security policies with their enforcement [1, 9, 17, 19, 21, 25]. Developers have to manually encode or call the enforcement mechanism to perform the security checks. This decreases code reusability and maintainability.

Specifying security policies with a DSL enables a decoupling between the specification language and the enforcement mechanism [29]. The security policy language then interacts with the enforcement mechanism by means of a well-defined interface that provides runtime information regarding a security-relevant operation. The only approach that provides decoupling is CoreScript, in which the developer has to provide the action that the enforcement mechanism needs to take for a given policy.

[1]For conciseness, we use the terms access control policy and security policy interchangeably in the rest of the paper.

2.5 Problem Statement

The previous observations have inspired the design of a novel approach for specifying and enforcing application-level access control policies, called Guardia. To the best of our knowledge, Guardia is the first approach to explore an internal DSL embedded in JavaScript for declaratively specifying security policies that features a decoupled enforcement mechanism without requiring VM modifications. Table 6 in Appendix A summarizes existing approaches and Guardia with respect to the analyzed design choices. More in detail, Guardia is the result of the following design decisions.

- The main design choice of our work is to explore language-based security that does not require VM modifications.
- Inspired by [6] and CoreScript [29], Guardia explores a domain-specific policy specification language.
- In contrast to those approaches, we explore an internal DSL embedded in JavaScript to express and compose complex policies. As both the target application and its security policies are written in the same language (JavaScript), this design choice may reduce the learning curve.
- A declarative specification of policies enables the decoupling between specification and enforcement. Guardia goes one step further than CoreScript and also allows developers to use different meta-programming APIs for the enforcement mechanism (e.g. JavaScript proxy API, Virtual Values, code instrumentation APIs [5, 24], etc.).

In this paper we more closely examine the consequences of the following design decisions: (1) choosing a declarative policy language as an internal DSL in JavaScript, and (2) employing an enforcement mechanism for access control security policies that solely relies on JavaScript proxies, so that VM modifications are not required. After introducing the Guardia policy specification language in the next section, we discuss the possibilities and limitations of only using reflection with respect to important security properties such as transparency, tamper-proofness, and completeness in Section 4.

3 DECLARATIVE SPECIFICATION OF SECURITY POLICIES USING AN INTERNAL DSL

In this section, we describe the specification language of Guardia that allows to declaratively express application-level access control security policies for client-side web applications written in JavaScript. Guardia provides a predefined set of fundamental policies that can be composed to build more complex ones. Fundamental policies alleviate the burden of correctly writing security policies, while the built-in composition mechanism provides the flexibility of imperative specifications.

Table 1 provides the overview of Guardia's policy specification API, which we detail in this section.

3.1 Attacker model

We assume that an attacker has found a way to bypass all security mechanisms provided by the browser (e.g., using unsanitized input) and was able to store JavaScript code in the application database as part of a user input mechanism (e.g., a user comments system). When a victim visits a page that loads and executes the attacker code as part of rendering that page, this attacker code is executed in the browser with the *same privileges* as the code of the page. Especially if the victim is an authenticated user of a sensitive application, the attacker is able to obtain sensitive information. In the same manner an attacker can cause application misbehavior by, for example, exhausting the application's resources.

3.2 Security policies in Guardia

In Guardia, a security policy is represented as an object that specifies a number of interception points used by the enforcement mechanism to monitor the application at runtime. This object defines two types of interception points that monitor security-relevant read and write operations, such as a method invocation or the assignment to a property in the target object, respectively. Developers can register listeners to monitor these read and write operations.

In contrast to WebJail, Guardia is not limited to a predefined set of components and objects on which the policies can be specified. Like CoreScript, developers can declaratively specify policies. However, Guardia's developers are still using JavaScript to specify the policies, while CoreScript ones have to switch to a different language (XML). Developers using Guardia do not have to write imperative *advice code* to implement the enforcement of the policies, because the advice code is implicit in the declaration of the policy.

Listing 1 shows a first sample policy in Guardia to monitor security-relevant read operations. More concretely, it shows an example policy definition that denies a read operation on the `open` method, which can be used to create new windows and get access to security sensitive methods [19]. The `whenRead` field takes an array of predicates that are evaluated on each read operation. A policy predicate (or simply a predicate) in Guardia is a closure that returns a boolean value, and is called by the enforcement mechanism to decide whether the actual call upholds the security invariant expressed by a policy. Similarly, the `readListener` field (line 4) takes an array of listeners that are notified on each read operation. Each registered listener is a JavaScript object that contains a `notify` function. The `notify` function (line 5) is executed each time a property is accessed or a method is invoked. This function receives as parameters the dynamic information related to the actual invocation. Unlike predicates, listeners do not return any value and their execution does not influence the enforcement.

Listing 1: Definition of a policy that denies a read operation on the `open` method.

```
1 const pol = {
2     whenRead: [Deny(['open'])],
3     whenWrite: [...],
4     readListeners: [{
5         notify: (tar, prop, rec, args) => {
6                 // update some state...
7         }}],
8     writeListeners: [...]
9 }
```

Besides `whenRead` and `readListeners`, Guardia also supports the dual write operations: `whenWrite` and `writeListeners`. In the remainder of this section, we introduce the different constructs in Guardia's

Construct	Description
`Allow(arr : Array) => TBase`	Allow the execution of the supplied properties
`Deny(arr : Array) => TBase`	Deny the execution of the supplied properties
`Not(p: TBase) => TBase`	Negates the result of the policy predicate given as parameter
`And(pArr: Array) => TBase`	Perform logical AND using predicates given as parameters
`Or(pArr: Array) => TBase`	Perform logical OR using predicates given as parameters
`ParamAt((...ps)=> Boolean, pIdx: Number, arr : Array) => TBase`	Apply a function to one parameter of the actual execution
`StateFnParam((...ps)=> Boolean,s: String, arr : Array) => TBase`	Apply a function to one state during an execution step
`getVType(idx: Number, fn : Function) => Object`	Returns an object in the following way fn(params[idx]), where params is injected by the enforcement mechanism.
`installPolicy(pol: Object) => Object`	Returns an object that deploys the policy
`on(tar: Object) => Object`	Returns a secured object

Table 1: Guardia's API

API for specifying and installing policies by means of examples from literature.

3.2.1 Policy 1: Prevent resource abuse. Client-side resource abuse in JavaScript can adversely affect user experience to the point that the application becomes unusable [19]. There exist certain methods in the DOM API that can be exploited for this kind of attack such as `prompt` and `alert` [4, 17, 19]. Listing 2 shows how to create a policy that prevents resource abuse of the methods `prompt`, `alert` and `confirm` in Guardia. At each invocation, the policy checks the name of the property being accessed. If the property is one of the property names specified by the policy, then the invocation is denied. To express this policy, we employ the `Deny` function which takes as argument a list of data and method properties that should be blocked upon access. Line 2 employs the `installPolicy` function to specify that the policy advice code should be executed when the user attempts to read a property upon the `window` object.

Listing 2: Policy 1: Prevent resource abuse.

```
1 let noResAbuse = Deny(['alert','prompt', 'confirm']);
2 installPolicy({whenRead:[noResAbuse]}).on(window);
```

Guardia also provides the `Allow` function which takes as parameter a list of properties that should not be blocked upon access.

`Allow` and `Deny` form the core primitives to build a simple policy in Guardia. Simple policies can be combined into more complex ones using the following *higher-order* policies predicates based on the three traditional logical operators:

- the `Not` function receives as parameter a policy predicate object A and returns a policy object predicate B that negates the behavior of A.
- the `And` function returns a predicate that evaluates to true if both of the predicates given as parameter return true.
- the `Or` function returns a predicate that evaluates to true if one of the predicates given as parameter returns true.

These higher-order policy predicates are crucial to be able to specify *control flow policies*. Control flow policies specify the control flow path that an execution should take. In what follows we specify two sample control flow policies from literature in Guardia.

3.2.2 Policy 2: Prevent dynamic creation of `iframe` *elements.* In this case, the execution of the `document.createElement(tag)` function must halt *only* when the value of the tag attribute is equal to `iframe`. As pointed out in Phung et al. [19], such a policy aims to solve attacks that can happen by restoring built-in methods from another page.

Listing 3 shows how to build a *no dynamic iframe creation* policy by negating the combination of a `Allow` and a `ParamAt` function. The `ParamAt` function returns a policy predicate that checks whether some property holds for specific parameter of a method invocation. In this example, `ParamAt` uses the function `equals` to ensure that the value of the tag passed as argument to function `createElement` is not equal to `'iframe'`. `ParamAt` primitive has three parameters:

- a predicate function that has two parameters;
- a function that safely extracts the value from the actual call argument, and passes that value to the predicate function;
- a value that is used by the predicate function.

In Listing 3 the `equals` function at line 1 is the predicate function. The function call `getVType(0, String)` at line 3 is intended to safely extract and use the call's arguments. The first argument, `0` in this example, represents the position of the argument in the call's arguments list. The second argument is a constructor that converts the extracted value, to a `String` in this case, ensuring that `equals` function will receive a string value as first argument. Line 4 deploys the policy on the document object to prevent the dynamic creation of `iframe` tags.

Listing 3: Policy 2: Prevent dynamic creation of `iframe`.

```
let equals = (a, b) => a === b
let notIframe = Not(And(Allow(['createElement']), ParamAt(equals,
                                    getVType(0, String), 'iframe')));
```

```
installPolicy({whenRead: notIframe}).on(document);
```

3.2.3 Policy 3: Limit number of popup windows. Kikuchi et al. [13] and Meyerovich et al.[17] define a policy to limit the number of attempts to open a popup window. This control-flow policy is actually a *stateful* policy that increments a counter each time a window is opened. Phung et al. [19] suggest that such a policy should also check that the new window has a location and status bar. We extended the invariant of this policy to also check that the URL is in a whitelist.

Listing 4 shows how to implement the resulting policy in GUARDIA. The policy specification verifies that the first parameter of the call to the `open` method is in a whitelist of URLs, and that the second parameter contains a location and status bar. The policy employs GUARDIA's `StateFnParam` primitive to assert upon some state of the application at a particular invocation as shown in line 22. Like `ParamAt`, this primitive should be combined with other primitives to limit an execution path to a certain behavior.

Listing 4: Policy 3: Limit number of popup windows.

```
1  var lstnr = {
2    notify: function (tar, name, rec, args) {
3      if (name === 'open') {
4        var winOpCnt = ac.getState('winOpenCount');
5          if (winOpCnt) {
6            winOpCnt += 1;
7            ac.setState('winOpenCount', winOpCnt);
8          } else {
9            ac.setState('winOpenCount', 1);
10         }
11       }
12     }
13 }
14 let contains = (a, b) => { return a.indexOf(b) != -1 }
15 let lessThan = (a, b) => { return a < b }
16 let limitWin =
17   Or(And(
18         Allow(['open']),
19         ParamInList(0,urls),
20         ParamAt(contains, getVType(1, String), 'location=yes'),
21         ParamAt(contains, getVType(1, String), 'status=yes'),
22         StateFnParam(le,'winOpenCount',3)),
23       Not(Allow(['open'])))
24
25 installPolicy({whenRead:[limitWin],
26               readListeners:[lstnr]}).on(window)
```

Examples shown in Listings 2 to 4 return closures intended to be used in the hooks (`whenRead` and `whenWrite`) shown in Listing 1. In GUARDIA, security policies must be deployed on objects. The result of deployment is an object containing the security properties that hold for all invocations on the target object. Listing 5 shows how to deploy a policy for the `window` object using the predicate in Listing 4.

Listing 5: Policy deployment example.

```
installPolicy({
  whenRead:[Not(And(Allow(['open']),
                    StateFnParam(lessThan,
                                 'popupCount',
                                 2)))]
}).on(window);
```

4 ENFORCEMENT OF SECURITY POLICIES USING JAVASCRIPT REFLECTION

To ensure that a target program satisfies a certain security policy, an enforcement mechanism is required that will notify or halt the system when the policy is violated [2, 23]. When a security policy is specified in the form of a predicate, the enforcement mechanism checks whether this predicate is true at each individual step of the program execution. Related work identifies the following desiderata for an enforcement mechanism [2, 4, 20].

(1) *Completeness*: an enforcement mechanism should be able to monitor and check all security-relevant operations that may be expressed by policies.
(2) *Transparency*: an enforcement mechanism should not alter the behavior of the program to be secured.
(3) *Tamper-proofness*: it should be impossible to subvert the enforcement mechanism itself.

Because GUARDIA's enforcement mechanism is decoupled from the policy specification API, this enables us to investigate different meta-programming techniques to enforce policies without VM modifications. In this paper, we explore an implementation of GUARDIA's enforcement mechanism employing the reflective capabilities of the host language itself (JavaScript).

4.1 Decoupled enforcement mechanism

GUARDIA decouples the specification of a policy from its enforcement. This forces a clear separation of these two concerns, and enables GUARDIA to be configured with different enforcement mechanisms. Figure 1 shows the interaction between different semantic blocks that make up GUARDIA.

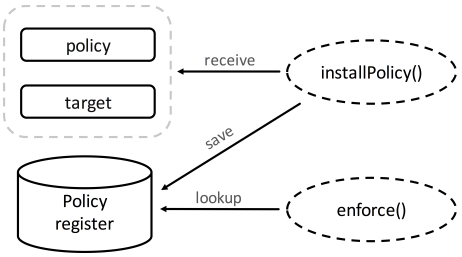


Figure 1: GUARDIA's policy deployment and enforcement process.

To better explain the decoupling from policy specification and enforcement, consider again the case that a programmer wants to prevent resource abuse via the DOM API. To this end, she designs a policy such as Listing 2 describing *what* should be protected (i.e., `alert`, `prompt`, etc.) and *when* the enforcement must be called (i.e., `whenRead`). Next, suppose the secured program reaches the call `window.alert('Foo')`. At this point, the enforcement will look for the policy configuration object associated with the target of the call (`window`). Then, the `filter(...)` method of all `whenRead` policy predicates is provided with the runtime information of the call. The runtime information includes the target (`window`), the method being read (`alert`), and the parameters (`'Foo'`). If all predicates return `true`, then the call is executed, otherwise an exception is thrown and

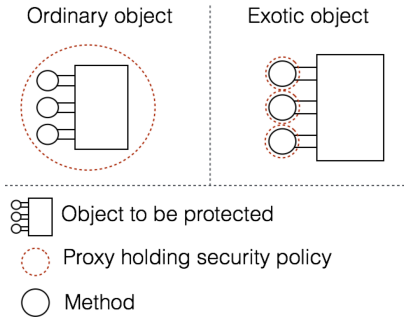


Figure 2: Wrapping enforcement approach in Guardia.

the call is not executed. Note that the test of the policy is encapsulated in the policy predicate and is not part of the mechanism that monitors the execution. Guardia can be configured with any enforcement mechanism that is able to monitor the execution and call `guardia.enforce(...)` with the appropriate runtime information whenever the application is about to read or write an object property (e.g. Virtual Values, Aran [2]).

4.2 Enforcement by means of JavaScript reflection

In this paper we focus on the default Guardia enforcement mechanism based on JavaScript's reflective capabilities using proxies [7]. A JavaScript proxy is an object that acts as a wrapper of another JavaScript object. By intercepting operations performed on the wrapped object, a proxy provides the means to change the semantics of those operations. A proxy object is created using the `Proxy(target, handler)` constructor, where `target` is the object to be wrapped and `handler` define various properties that enable behavioral intercession.

In order for the enforcement mechanism to adhere to the completeness requirement, policies should be deployable on all types of Javascript objects. However, browser host environments provide *exotic objects* such as `window`, `document`, `location`, etc. Exotic objects differ from ordinary objects in that they do not implement the default behavior for one or more of the essential internal methods that must be supported by all objects [7]. This adds extra difficulties to an enforcement mechanism based on proxies, as exotic objects require different monitoring strategies for policy enforcement. In what follows, we detail the enforcement using proxies upon both ordinary and exotic JavaScript objects.

4.2.1 Enforcement upon ordinary JavaScript objects. A reference to an ordinary (i.e., non-exotic) object can be freely reassigned with a proxy that secures it. This is shown on the left hand side of Figure 2. To prevent problems with aliasing and ensure that attackers only have access to the secured version of a sensitive object, the enforcement mechanism must secure objects right after their creation.

Listing 6 illustrates how object proxy handlers are implemented in Guardia. Of particular interest are the `get` and `set` properties, which reify the semantics of how object properties should be read and written, respectively.

Listing 6: Guardia proxy handler's implementation.

```
1  let handler = {
2    get: function (trgt, prop, rcvr) {
3         ...
4         for(let pol of policy['whenRead']){
5             if(!pol.filter(trgt, prop, rcvr, 'propertyRead')){
6               throw new Error('Not allowed!')}}}
7         ...
8         notify(policy['readListeners'], trgt, prop);
9         return Reflect.get(trgt, prop, rcvr);
10   },
11   set: function(trgt, prop, value, rcvr){
12        ...
13   }
14 };
15
16 let target = new Proxy(trgt, handler);
```

Whenever a property read occurs on a secured object, the proxy intercepts this and forwards the operation to the `get` method of its handler. Lines 4–9 specify the semantics of Guardia for verifying whether the property read is allowed. First, Guardia iterates over each policy predicate contained in the `whenRead` property of the policy configuration object (line 4). The method `filter` is called on each predicate with the runtime information provided in the actual call, which determines whether the call violates the policy or not (line 5). If any policy is violated, the handler throws by default an exception, thereby preventing the actual read operation on the underlying secured object (line 6). Otherwise, all the registered listeners are notified (line 8) and the read operation on the underlying object is performed (line 9). A similar approach holds for property write operations, which are intercepted by the `set` method on the handler.

4.2.2 Enforcement upon exotic objects. Exotic objects pose a challenge to a reflective enforcement approach because they are read-only references according to the HTML5 standard [28]. Developers are able to modify these objects by adding or deleting properties, but it is forbidden to change their references[3].

Instead of wrapping the entire object as in the case of ordinary objects, it is necessary to wrap each *method* of the exotic object with a proxy enforcing the relevant security policies. This is shown on the right hand side of Figure 2. This approach respects the invariants of the exotic object, while still introducing the necessary checks on security-sensitive operations on those objects.

To illustrate this approach in a concrete example, we take Policy 2 (Section 3.2.2), which prevents dynamic creation of `iframe` objects by disallowing the call expression `document.createElement('iframe')`. Instead of wrapping the entire `document` object, Guardia only wraps the `document.createElement` function object as shown in Listing 7. The handler has to intercept a function invocation, and therefore implements an `apply` operation.

Listing 7: Function proxy handler's implementation.

```
1  let handler = {
2    apply : function (target, thisArg, argumentsList){
3              //Check the security policies
4              return Reflect.apply(target, thisArg, argumentsList)}
```

[2] https://github.com/lachrist/aran

[3] An exception to this rule is the `location` object that, when assigned with a location, causes the browser to navigate to that location.

```
5  };
6
7  Object.defineProperty(document, 'createElement', {
8      configurable: false,
9      writable: false,
10     value: new Proxy(document.createElement, handler)}
11 );
```

Line 3 in Listing 7 is a placeholder for the enforcement code that verifies whether the function call is allowed. Checking the predicates contained in the relevant policy configuration objects uses a similar mechanism as in Section 4.2.1. The method `Object.defineProperty` enables the addition or modification of a property on an object. Lines 7–10 replace the original `document.createElement` function object with the wrapped one on `document`. The properties `configurable` and `writable` are set to `false` to prevent any subsequent modification of the `document.createElement` property.

4.3 Limitations and discussion

Using only proxies as the basis for a policy enforcement mechanism has an impact on completeness, transparency, and tamper-proofness of the resulting mechanism. In the following subsections, we discuss how well Guardia's enforcement mechanism described in Section 4.2 achieves these properties, and point out some of the challenges such a reflection-based enforcement strategy introduces.

4.3.1 Completeness. Guardia's proxy enforcement mechanism does not require any modification of the underlying JavaScript runtime, but it is not fully complete. This is because of the `location` object, an exotic DOM object that is non-configurable, including its methods. It is impossible to wrap the `location` object with proxies that intercept security-relevant operations such as changing the location by invoking `location.assign`.

4.3.2 Transparency. The goal of Guardia's proxy enforcement mechanism is to achieve transparency w.r.t. the original (unsecured) application by ensuring that the behavior of wrapped target objects remain unaffected. To this end, we conducted experiments to investigate how proxies behave in real-world applications on different browsers when using popular libraries such as JQuery. First experiments revealed some issues. In particular, JQuery presented errors when methods on the `window` or `document` objects were wrapped. Further investigation showed that JQuery uses the `toString` function of methods on host objects to assert whether the containing host objects are native or not. However, this check fails when proxies wrap these functions. Guardia overcomes this problem by binding the wrapped `toString` function to the target object instance instead of the proxy.

Our experiments also revealed that proxies do not behave transparently on DOM `Node` objects. The `node.appendChild(child)` function, for example, checks that the argument value is of type `Node`. When this method receives a proxy, the type check fails and the node is not added to the tree. To overcome this problem, Guardia handles `Node` instances as exotic objects: instead of wrapping the entire object, every function on the object is wrapped.

4.3.3 Tamper-proofness. Making Guardia itself secure is challenging in JavaScript, especially when the specification is done using an internal DSL and the host language's reflective capabilities are employed as the basis of the enforcement mechanism. Javascript is a prototype-based language, and therefore attackers can attempt to change the behavior of the system by altering the prototype chain of objects that make up or participate in the security mechanism. In the remainder of this section we discuss three attacks that can compromise the tamper-proofness of Guardia's enforcement based on proxies: (1) redefinition of `toString` and `valueOf` functions, (2) function aliasing, and (3) prototype poisoning.

Redefinition of `toString` *and* `valueOf` *functions.* Listing 8 shows a code snippet that demonstrates how an attacker could provide an object that redefines the `toString` function. The first invocation of this function returns a 'good' URL, but subsequent invocations return a 'bad' URL provided by the attacker. If the `liarObj` object is used during policy evaluation to verify whether a URL is whitelisted, the whitelist policy can be bypassed.

Listing 8: Example of `toString` redefinition.

```
var liarObj = {
       value : 'good',
       toString : function() {
              var result = this.value;
              this.value = 'bad';
              return result;
       }
}
console.log(liarObj.toString()) // good
console.log(liarObj.toString()) // bad
```

To avoid this problem, Guardia adopts the same approach as Magazinius et al. [15] and converts all policy parameters to primitive values once, and only uses the converted values in subsequent target invocations.

Function aliasing. Guardia's policy specification language relies on the names of functions and properties to validate their invocation. Relying on names to ensure security is a straightforward way to restrict access to certain data or functionality. However, in JavaScript, it is easy to create function aliases because functions are first-class objects allocated on the heap. For example, the `window.open` function can be aliased with a function `myFun` by assignment: `myFun = window.open`. An attacker could then use the aliased function to circumvent the security policy enforcement mechanism [17, 19].

To prevent the risks associated with aliasing, the deployment of security policies must be realized *before* any other code is executed that can create aliases of target objects. If this is the case, then all aliases that are created refer to the secured object so that the underlying target object is never exposed to client code. Additionally, Guardia freezes wrapped methods by means of calling `Object.freeze` on them. Freezing wrapped functions avoids the aliasing problem by preventing method overriding.

Prototype poisoning. An attacker could take advantage of an object's prototype inheritance chain to compromise Guardia's tamper-proofness. Because every JavaScript object is created in an extensible and configurable state, properties can be freely added and modified at any point during the object's lifetime. An attacker could therefore attempt to subvert the execution of the target program by changing the prototype of a built-in (e.g., `Object`, `String`, `Array`, etc.) or policy object, with the goal of abusing the inheritance chain to inject an alternative implementation of some method to bypass a security

policy. This type of attack is referred to as *object subversion* [15]. For example, callers of the `Object.prototype.toString` function always expect a string representation of the object on which it is invoked. An attacker could inject a `toString` function similar to the one shown in Listing 8 for compromising the security of the application.

Using ECMAScript 5 property descriptors, an object can be marked as *non-extensible* so that it is not possible to add new properties to the object after creation, or *non-configurable* so that any attempt to change its non-configurable properties fails. In order to prevent unintended changes to Guardia's enforcement constructs, Guardia makes use of these descriptors in its implementation. The elements being frozen after their creation include all the objects that are involved in the definition of and interaction with Guardia's API (Table 1), and standard objects. These deliberately imposed constraints on the prototype chain of built-in objects such as `Object` and `String` could affect the transparency of programs that rely on changing the prototype of those objects. However, during our experiments (Section 5) we encountered few problems as a result of this strategy (see Section 5.2.3).

5 EVALUATION

To evaluate Guardia with respect to the design decisions detailed in Section 2, we conducted three kinds of experiments. In a first experiment, we expressed 13 different security policies in Guardia extracted from literature (Section 3 and Appendix B). This enables us to compare the expressivity of Guardia to that of related approaches (Section 5.1). A second experiment consisted of applying Guardia to both synthetic benchmarks, three experimental web applications, and 10 real-world web applications. Finally, we conducted a third experiment to evaluate the performance implications of our approach on both synthetic benchmarks and the experimental applications (Section 5.3).

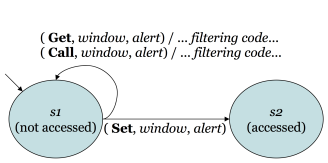
5.1 Expressivity Compared To Related Work

We evaluated the expressiveness of Guardia's specification language by expressing 13 policies found in related work [9, 17, 19, 29]. Table 2 gives an overview of these policies and their origin. A checkmark denotes that a paper describes and supports the policy, while a missing checkmark does not imply that a paper does not support a policy but rather that it does not describe the policy. Table 2 extends the table presented in [4] with the type of attack that each policy aims to prevent. In contrast to the original table, we consider only 11 distinct policies (denoted as Policy 1–11) because several policies could be combined into a single policy.

Table 2 shows that all resulting 11 policies analyzed in related work can be expressed in Guardia. For each policy, we compared the specification in Guardia with the specification in related work. Due to space limitations, we report on this comparison for only 4 out of the 11 policies below. Appendix B includes the implementation of the 7 remaining policies in Guardia.

5.1.1 CoreScript. We compare Policy 1 specified in CoreScript (Figure 3) to the specification in Guardia (Listing 2). This policy prevents resource abuse by denying the creation of `alert` windows.

Originally, CoreScript security policies were described using a formalism based on edit automata [29], but in a follow-up paper [13] developers can also encode policies by writing XML files. The XML




```
<template name = "FuncReplacement">
  <object> obj </object>
  <property> prop </property>
  <states> s1, s2 </states>
  <replacementAction>
    ... filtering code ...
  < /replacementAction>
</template>
```

Figure 3: (Policy 1) Prevent resource abuse in CoreScript [29].

in Figure 3) specifies Policy 1. In CoreScript, developers identify the object, property, or method to which code rewriting must be applied. Instead of forcing the developer to think in term of state and transitions, which may not be common knowledge among developers and security engineers, Guardia uses a declarative and arguably more descriptive approach for specifying security policies.

In contrast to Guardia, CoreScript forces developers to know how to write the *replacement action* code. The replacement action should perform the actual enforcement of the policy. In our view it is less error-prone to specify a policy that prevents certain behavior than to manually write code that should behave similar to the replaced code, while at the same time taking care of the enforcement and transparency with regard to normal program execution. In Guardia developers are not burdened with writing enforcement code. The semantics of the operations and their properties, such as transparency and tamper-proofness (with the limitations we discussed), are provided by the underlying enforcement used by Guardia, and therefore well understood.

5.1.2 ConScript. Listing 3 (Section 3) introduced Policy 2 in Guardia to prevent dynamic `iframe` creation. We compare this policy to the equivalent ConScript policy specification [17] given in Listing 9. As mentioned in Section 2, ConScript specification follows an aspect-oriented approach in which a pointcut is declared to intercept relevant calls, in this case to `document.createElement`. ConScript forces programmers to write code for *both* policy specification and enforcement in the language of the VM. As a result, programmers have to manually ensure the completeness, transparency, and tamper-proofness of the enforcement mechanism. In contrast, Guardia developers only have to declare security policies without programming their enforcement.

Listing 9: (Policy 2) Prevent dynamic creation of `iframe` in ConScript (extracted from [17]).

```
1 around(document.createElement, function (c : K, tag : U) {
2 let elt : U = uCall(document, c, tag);
3 if (elt.nodeName == "IFRAME") throw 'err'; else return elt; });
```

ConScript relies on VM modifications and can only be applied to Internet Explorer 8, while Guardia runs in any browser that implements the ECMAScript 2015 standard. On the other hand, Guardia's specification of Policy 2 is slightly more verbose than its ConScript equivalent.

5.1.3 Hallaraker and Vigna's Auditing System for JavaScript. Listing 10 shows Policy 3 specified in Hallaraker and Vigna's auditing system for JavaScript (HVAS) [9]. This policy limits the number of popups that a window can open. The equivalent Guardia policy specification was given in Listing 4 (Section 3). Policies in HVAS

Attack type	Security policy	HV [9]	Yu et al. [29]	Phung et al. [19]	ML [17]	GUARDIA
Forgery	Limited number of popup windows opened (Policy 3)	✓	✓	✓	✓	✓
Forgery	No popup windows without location and status bar (Policy 3)			✓		✓
Resource abuse	Prevent abuse of resources like modal dialogues (Policy 1)	✓	✓	✓	✓	✓
Restoring built-ins from frames	Disallow dynamic iframe creation (Policy 2)			✓	✓	✓
Information leakage	Disable page redirects after `document.cookie` read (Policy 6)	✓	✓	✓	✓	✓
Information leakage	Only redirect to whitelisted URLs (Policy 10)			✓	✓	✓
Information leakage	Restrict `XMLHttpRequest` to secure connections and whitelist URLs (Policy 9)				✓	✓
Information leakage	Disallow setting of `src` property of dynamic images (Policy 11)			✓		✓
Impersonation	`XMLHttpRequest` is restricted to HTTPS connections (Policy 9)				✓	✓
Impersonation / Information leakage	Disallow open and send methods of XHR object (Policy 4)			✓	✓	✓
Man in the middle	`postMessage` can only send to the origins in a whitelist (Policy 7)				✓	✓
Run arbitrary code	Disallow `string` arguments to `setInterval` & `setTimeout` (Policy 8)				✓	✓
Information Leakage	Disable geoposition API (Policy 5)					✓

Table 2: Comparison of approaches in security policies. Policy numbers 1–11 refer to the policies discussed in Sections 3 and 5 and Appendix B

are specified as a state transition model. While both specifications are expressed in more or less the same amount of code, the number of allowed popup windows is hardwired in the HVAS specification. In HVAS, the policy designer has to write as many `if` statements as the number of popups that are allowed, which hampers code maintainability and reusability. In contrast, GUARDIA's specification parametrizes the maximum allowed number of popup windows as an argument of the policy.

Listing 10: (Policy 3) Limit number of popup windows in HVAS (extracted from [9]).

```
if((event.method.name==open)&&
    (event.method.object=="window")){
if(stateW4.includes(event.host)){
   log("Script has opened 5 windows. Possibly a malicious script!")
}
else if(stateW3.includes(event.host)){
    stateW3.delete(event.host);
    stateW4.add(event.host);
  }
else if(stateW2.includes(event.host)){
    stateW2.delete(event.host);
    stateW3.add(event.host);
}
else if(stateW1.includes(event.host)){
    stateW1.delete(event.host);
    stateW2.add(event.host);
  }
  else{
    stateW1.add(event.host);
} }
```

Furthermore, the GUARDIA specification makes it straightforward to add and combine additional policy predicates for imposing additional restrictions as part of the policy. Recall that the code in Listing 4 also restricts the URLs that can be opened (line 18), and that the location and status bar of the newly opened windows must be visible (lines 19–20).

5.1.4 Lightweight Self-Protecting JavaScript. We compare Lightweight Self-Protecting JavaScript (LWSPJS) [19] to GUARDIA by means of Policy 4 that prevents impersonation attacks using the `XMLHttpRequest` (XHR) object by disallowing calls to its `open` and `send` methods.

Listing 11: (Policy 4) Prevention of impersonation attacks in LWSPJS (extracted from [19]).

```
1
2  var XMLHttpRequestURL = null;
3  enforcePolicy ({ target : XMLHttpRequest , method: 'open' },
4      function(invocation){
```

```
      XMLHttpRequestURL = stringOf(invocation ,1);
      return invocation.proceed();
    });

enforcePolicy({ target : XMLHttpRequest , method: 'send'},
   function(invocation){
   XMLHttpRequestPolicy(invocation);
});

var XMLHttpRequestPolicy = function(invocation){
    //allow the transaction if the URI is in the whitelist
    if (AllowedURL(XMLHttpRequestURL))
       return invocation.proceed () ;
    policylog('XMLHttpRequest_is_suppressed:'+
     'potential_impersonation_attacks') ;
}
```

Listing 11 shows the specification of Policy 4 in LWSPJS, in which the URL passed to the `open` method is forced to be a `String`. The policy deployed upon the `send` method verifies that the URL string is contained in the whitelist of URLs. Developers have to manually specify the enforcement code (lines 3–7, 9 –11, and 13–18) and consequently most of the code in Listing 11 is dedicated to the enforcement.

Listing 12 shows the equivalent code for Policy 4 in Guardia, which requires less code than LWSPJS to express the same policy, while the intention of the policy is still explicit. This is because Guardia does not require developers to manually write the enforcement code.

Listing 12: (Policy 4) Prevention of impersonation attacks in Guardia.

```
const openPol = Or(And(Allow(['open']),
                       ParamAt(isIn, getVType(0, String), whiteList)),
                   Deny(['open']));
XMLHttpRequest = installPolicyCons(openPol, XMLHttpRequest);
```

5.2 Applicability

To assess the applicability of a reflection-based policy enforcement, we used Guardia's enforcement mechanism based on proxies to secure three types of programs: small synthetic benchmarks, experimental web applications, and real-world web sites.

Listing 13 shows how developers can include the necessary files needed to secure their application with Guardia. First, the implementation file (`guardia.js`) containing Guardia's constructs must be included. Additionally, developers include a file (typically called `policies.js`) that contains any number of application-specific Guardia policies such as those discussed in this paper (see Table 2). Besides including the required files, and depending on the type of application, other small changes may be required to deploy Guardia. For Single Page Applications, including Guardia in the initial page suffices to secure the entire application. For sites that reload the browser window for each request, Guardia can be added by using a proxy mechanism in the server that modifies each response.

Listing 13: Guardia policy deployment example.

```
<html>
  <head>
    <script src="path/to/guardia.js"></script>
    <script src="path/to/policies.js"></script>
  </head>
  <body> ... </body>
</html>
```

5.2.1 Correctness on synthetic benchmarks. A suite of synthetic benchmarks was used to drive forward the implementation of Guardia by testing new functionality and avoiding regressions. Each program in the set of synthetic benchmarks is implemented in such a way that it is straightforward to determine whether a vulnerability (or some other kind of behavior) is present or absent. We then developed Guardia policies targeting these benchmarks and verified for each synthetic benchmark whether the results of policy enforcement agreed with the expectations. For more details on the suite of synthetic benchmarks, we refer the interested reader to the publicly available implementation of the Guardia framework[4], which contains this test suite.

5.2.2 Practicality and transparency. Guardia was tested on three experimental applications: Juice Shop, NodeGoat and SoundRedux. Juice Shop and NodeGoat are part of the Open Web Application Security Project (OWASP) project, which serves as learning resource for application security. By design, both applications have security holes that can be used by developers and penetration testers to learn how to protect their applications. SoundRedux provides a fully functional application in a complex scenario. Because all three applications use contemporary JavasScript libraries and frameworks, securing them with Guardia provides a good notion of how practical our approach is. It also enables us to assess the transparency of Guardia's enforcement mechanism based on proxies in real-world scenarios.

OWASP Juice Shop. Juice Shop[5] is a typical online shopping application with search, listing, and shopping basket functionalities, in which users are required to register and login. Juice Shop has been intentionally designed to include the entire OWASP Top Ten vulnerabilities and other security flaws. It is developed entirely using JavaScript technologies in both the back-end and the front-end. Its front-end technologies includes JQuery, AngularJS, and Twitter Bootstrap.

As mentioned before, Guardia is implemented as a JavaScript library and can therefore be deployed in any standard ECMAScript 5 (or more recent) runtime environment, including web contexts, using standard mechanisms. Juice Shop is a Single Page Application, so Guardia must only be included once in this application.

We applied Guardia's implementation of the policies described in Table 2 to Juice Shop to protect the application from Reflected Cross Site Scripting attacks [18, 26]. We found that we were able to enforce all policies except Policy 10, which targets the `location` object. As explained in Section 4.2.2, the `location` object imposes strong invariants that makes it impossible to protect it without relying on VM modification.

OWASP NodeGoat. NodeGoat [6] is a vulnerable web application that manages employee retirement savings. The application offers typical functionalities such as user login and registration. Registered users have a private dashboard page in which they can modify their preferences and manage their benefits.

[4] https://github.com/scull06/guardia
[5] https://github.com/bkimminich/juice-shop
[6] https://github.com/OWASP/NodeGoat

NodeGoat has similar security vulnerabilities as those found in Juice Shop. It is developed using current technologies and includes libraries such as JQuery and Twitter Bootstrap. We therefore applied the same set of security policies to NodeGoat as to Juice Shop and obtained the same results in terms of security.

SoundRedux. SoundRedux [7] is a client-side web application that serves as an interface to the SoundCloud [8] application, which enables exploring the SoundCloud music database. In contrast to NodeGoat and Juice Shop, SoundRedux is not a deliberately insecure web application, and is fully functional instead.

SoundRedux is developed using popular software libraries such as React [9] and Redux [10]. To deploy Guardia in SoundRedux, we modified its index page by adding a `script` tag for including Guardia itself, and a second one pointing to our set of security policies.

In contrast to the previous two applications, we did not perform any kind of attack on SoundRedux through its interface, as the application does not have any obvious security breaches and it is not the aim of this paper to *discover* security holes. Instead, we found that deployed policies were fully and correctly enforced by running code in the browser's developer console that attempts to bypass the deployed policies. We also verified that safe code was unaffected, showing that Guardia's behavior is transparent with respect to the SoundRedux application.

5.2.3 Transparency on web applications. In another experiment we applied our set of Guardia policies (Table 2) to 10 real world web applications (Table 3) to verify that these web sites continue to perform as expected in the presence of Guardia. The selection of the applications is based on the Alexa top 500 ranking[11], from which we selected the sites based on their purpose (i.e. *news, shopping, entertainment, social network, etc.*). Although the web sites vary in their intended use, all involve substantial amounts of complex JavaScript code that runs in the browser.

We employed the Burp Suite [12] to deploy our policies in these applications. Burp enables to intercept responses from these web sites and to inject Guardia policies. As a result, when the page is rendered in the browser it contains the deployed policies. Because the applications listed in Table 3 do not have evident security holes, we again tested the policies of Table 2 by writing code in the browser's console attempting to bypass these policies.

The result of the experiment was that all sites, except *YouTube*, continued to function as designed in the presence of Guardia. Closer inspection revealed that *YouTube* attempts to override properties that were secured and sealed by Guardia policies. The *Vimeo*, *eBay*, *Reddit* and *BBC* web sites also did not render correctly at first. Inspecting the produced error trace indicated that these applications were attempting to create `iframe` elements dynamically and that Guardia was preventing this behavior. These web sites executed normally after removing Policy 2, which disallows the dynamic creation of `iframe` elements.

[7] https://github.com/andrewngu/sound-redux
[8] https://soundcloud.com/
[9] https://facebook.github.io/react/
[10] http://redux.js.org/
[11] https://www.alexa.com/topsites
[12] https://portswigger.net/

Application	Type	Deployed
google.com	Search Engine	✓
baidu.com	Search Engine	✓
bbc.com	News Site	✓
reddit.com	News Site	✓
youtube.com	Entertainment	
vimeo.com	Entertainment	✓
amazon.com	Online Shopping	✓
taobao.com	Online Shopping	✓
ebay.com	Online Shopping	✓
linkedin.com	Social Network	✓

Table 3: Real-world applications tested with Guardia.

5.3 Performance

To assess Guardia's performance impact, we measured the runtime overhead of deploying Guardia policies in the three types of benchmark programs we experiment with: small synthetic benchmarks, experimental web applications, and real-world web sites. These experiments were performed on a MacBook Pro with a 2.5 GHz Intel Core i7 processor equipped with 16 GB of DDR3 RAM.

Policy	`document.createElement()`	`document.write()`	`window.setTimeout()`	`window.setInterval()`
Simple Predicate	1.36x	1.13x	1.22x	1.22x
Simple + Combined Predicate	1.92x	1.28x	1.33x	1.31x
10 Simple Predicates	3.67x	1.78x	1.50x	1.42x

Table 4: Overhead of Guardia on synthetic benchmarks.

5.3.1 Performance on synthetic benchmarks. Table 4 shows the overhead introduced by Guardia on synthetic benchmarks that call a particular function using different policy constructs.

- Simple Predicate is a policy that enforces a single predicate (e.g. `Deny(['write'])`).
- Combined Predicate is a policy that enforces a single predicate using policy combinators (e.g. `Not(Allow(['write']))`).

To measure the overhead we ran the program 100000 times for every combination of policy construct and function. Each row of the table indicates the slowdown ratio between the average overhead and the average baseline. The average overhead of our worst case scenario is a 2.01x slowdown.

Application	Description	LOC	Load time (ms)	Load time with Guardia (ms)	Overhead	polchecks
Juice Shop	Online Shop	39220	36.65	42.22	1.15x	56
NodeGoat	Social Security App	16455	222.05	239.32	1.08x	25
SoundRedux	Soundcloud Client	48880	103.43	277.95	2.69x	71

Table 5: Experimental applications tested with Guardia.

5.3.2 Performance on experimental applications. We measured the performance impact of using Guardia to deploy Policy 2 and Policy 8 in Juice Shop, NodeGoat, and SoundRedux. Other policies were either not triggered (e.g., Policies 6, 7, 11), or were difficult to measure because they require user interaction to open or close popups windows (e.g., Policy 1 and Policy 3).

Each application was loaded 100 times to determine the average load time. The time spent by the browser to load the main document was measured by computing time differences using `performance.now()`. Table 5 relates the lines of JavaScript Code (LOC), the page load time without and with Guardia, and the overhead provoked by Guardia. *polchecks* is the number of calls to the policy check in each request.

From the results in Table 5 we conclude that there is negligible overhead when enforcing Policy 2 and Policy 8 during each page load. Although the policy checks are triggered several times in each application, this does not significantly impact the performance of those applications.

5.3.3 Performance on real-world applications. We attempted to measure the performance overhead introduced by Guardia on the applications listed in Table 3. To this end we used *mitmproxy* [13] to cache the responses of applications. Next, we recorded page load times with and without Guardia policies. However, we found that the performance impact introduced by Guardia is negligible compared to the variance introduced by the amount of resources (images, scripts, styles, etc.) loaded by these applications. Which made impossible to measure the performance impact introduced by Guardia.

5.4 Extensibility

Although Guardia was developed primarily for securing client-side applications, both its specification language and enforcement mechanisms can be extended to other application domains and runtime environments that feature objects and functions.

Nothing prevents our current implementation of Guardia to be used in server-side JavaScript applications. There it can be used, for example, to safely exchange valuable resources such as database connection objects with untrusted code by only allowing `read` operations.

Guardia facilitates extensibility by decoupling specification from enforcement. Dynamic enforcement of Guardia policies depends on the meta-programming facilities present in the underlying runtime environment. We believe that enforcement through code writing is always a viable option, even in the absence of advanced reflective capabilities in the target language (Section 6).

[13] https://mitmproxy.org/

6 CONCLUSION

In this paper, we presented Guardia, an internal DSL for declaratively specifying and dynamically enforcing application-level security policies for JavaScript web applications without requiring VM modifications. Guardia enables the specification of composable security policies that combines the flexibility of imperative specification languages with the ease of development provided by more declarative solutions. To evaluate our declarative policy specification language, we implemented 13 access control security policies from related work and found that Guardia's specification language is capable of expressing all of them.

Security policies in our approach are enforced by the underlying enforcement mechanism, which is decoupled from the specification language, which frees developers from manually enforcing security policies. In this paper, we focus on the default Guardia enforcement mechanism that employs JavaScript's reflective capabilities. This mechanism wraps target objects with proxies that intercept security-relevant invocations, and therefore does not require VM modifications. We discussed the limitations of this reflection-based enforcement mechanism with respect to completeness, transparency, and tamper-proofness.

We also evaluated the applicability and performance impact of our dynamic enforcement mechanism in three experimental applications and 10 real-world web sites. Our experiments indicate that the reflection-based enforcement mechanism of Guardia is correct, transparent, and tamper-proof, while incurring a reasonable runtime overhead. We believe the lessons learned from our study can be used by other application-level security policy approaches, as well as to stir future improvements on JavaScript VM.

Future Research Avenues. We are experimenting with a different enforcement mechanism for Guardia that combines enforcement based on proxies with *code rewriting* to provide a higher level of completeness than the enforcement described in this paper. This enables Guardia to be used for enforcing *information flow* policies, which cannot be covered by an enforcement mechanism that solely relies on the built-in reflective capabilities of JavaScript. Code rewriting mechanisms also enable a more uniform reasoning about a program's value properties, which alleviates some of the tamper-proofness challenges we needed to solve when using only reflection-based policy enforcement. Repeating our experiments using enforcement by code rewriting is ongoing work.

REFERENCES

[1] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: Complete Client-side Sandboxing of Third-party JavaScript Without Browser Modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2420950.2420952

[2] James P. Anderson. 1972. *Computer Security Technology Planning Study.* Technical Report ESD-TR-73-51. U.S. Air Force Electronic Systems Division.

[3] Thomas H Austin, Tim Disney, Cormac Flanagan, Thomas H Austin, Tim Disney, and Cormac Flanagan. 2011. *Virtual values for language extension.* Vol. 46. ACM.

[4] Nataliia Bielova. 2013. Survey on JavaScript security policies and their enforcement mechanisms in a web browser. *The Journal of Logic and Algebraic Programming* 82, 8 (Nov. 2013), 243–262.

[5] Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, and Coen De Roover. 2016. Linvail - A General-Purpose Platform for Shadow Execution of JavaScript. *SANER* (2016), 260–270.

[6] Sophia Drossopoulou, James Noble, and Mark S. Miller. 2015. Swapsies on the Internet: First Steps Towards Reasoning About Risk and Trust in an Open World. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security (PLAS'15).* ACM, New York, NY, USA, 2–15. https://doi.org/10.1145/2786558.2786564

[7] Ecma International. 2015. *ECMAScript 2015 Language Specification* (6th ed.). Ecma International, Geneva. http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf

[8] D Ghosh. 2011. *DSLs in Action.* Manning. 351 pages.

[9] Hallaraker, O and Vigna, G. 2005. *Detecting malicious JavaScript code in Mozilla.* IEEE.

[10] Kevin W Hamlen, Micah Jones, and Meera Sridhar. 2012. Aspect-Oriented Runtime Monitor Certification. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings.* Springer Berlin Heidelberg, Berlin, Heidelberg, 126–140.

[11] Xing Jin, Tongbo Luo, Derek G. Tsui, and Wenliang Du. 2014. Code Injection Attacks on HTML5-based Mobile Apps. *CoRR* abs/1410.7756 (2014). http://arxiv.org/abs/1410.7756

[12] Micah Jones and Kevin W Hamlen. 2010. Disambiguating aspect-oriented security policies. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development, AOSD 2010, Rennes and Saint-Malo, France, March 15-19, 2010.* ACM Press, New York, New York, USA, 193–204.

[13] Haruka Kikuchi, Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. 2008. JavaScript Instrumentation in Practice. In *Programming Languages and Systems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 326–341.

[14] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. 2015. The Unexpected Dangers of Dynamic JavaScript. In *24th USENIX Security Symposium (USENIX Security 15).* USENIX Association, Washington, D.C., 723–735. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lekies

[15] Jonas Magazinius, Phu H Phung, and David Sands. 2012. Safe Wrappers and Sane Policies for Self Protecting JavaScript. In *Informatics.* Springer Berlin Heidelberg, Berlin, Heidelberg, 239–255.

[16] Leo A Meyerovich, Adrienne Porter Felt, and Mark S Miller. 2010. Object views: Fine-Grained Sharing in Browsers. In *the 19th international conference.* ACM Press, New York, New York, USA, 721–730.

[17] Leo A Meyerovich and Benjamin Livshits. 2010. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *2010 IEEE Symposium on Security and Privacy.* IEEE, 481–496.

[18] G K Pannu. 2014. A Survey on Web Application Attacks. *IJCSIT) International Journal of Computer Science and ...* (2014).

[19] Phu H. Phung, David Sands, and Andrey Chudnov. 2009. Lightweight Self-protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (ASIACCS '09).* ACM, New York, NY, USA, 47–60. https://doi.org/10.1145/1533057.1533067

[20] Charles Reis, John Dunagan, Helen J Wang, Opher Dubrovsky, and Saher Esmeir. 2007. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Transactions on the Web (TWEB)* 1, 3 (Sept. 2007), 11–es.

[21] Gregor Richards, Christian Hammer, Francesco Zappa Nardelli, Suresh Jagannathan, and Jan Vitek. 2013. Flexible Access Control for Javascript. *SIGPLAN Not.* 48, 10 (Oct. 2013), 305–322. https://doi.org/10.1145/2544173.2509542

[22] H Saiedian and D Broyle. 2011. Security vulnerabilities in the same-origin policy: Implications and alternatives. *Computer* (2011).

[23] Fred B Schneider. 2000. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)* 3, 1 (Feb. 2000), 30–50.

[24] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of JavaScript. In *... Joint Meeting on Foundations of ...*. ACM, 615–618.

[25] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. 2011. WebJail: least-privilege integration of third-party components in web mashups. *ACSAC* (2011), 307–316.

[26] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS07.*

[27] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. 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.* ACM Press, New York, New York, USA, 1376–1387.

[28] WHATWG. 1017. *HTML Standard.* html.spec.whatwg.org.

[29] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. 2007. JavaScript instrumentation for browser security. *ACM SIGPLAN Notices* 42, 1 (Jan. 2007), 237–249.

A RELATED WORK SURVEY

Table 6 summarizes the existing solutions for specifying and enforcing access control security policies surveyed in Section 2 with respect to the design choices identified.

B ADDITIONAL SECURITY POLICIES

B.0.1 Policy 5: Disable geoposition API. Geo-location API allows to gather the physical location of the device. In spite of that browsers have a policy that asks user explicitly for using the geolocation information, it is desirable to deactivate the use of this feature programmatically.

Listing 14: Policy 5: Disable geoposition API in GUARDIA.

```
G.installPolicy({
    whenRead: [G.Deny(['getCurrentPosition','watchPosition',
                       'clearWatch'])]
}).on(navigator.geolocation)
```

B.0.2 Policy 6: Disable page redirects after `document.cookie` *read.* Cookies are commonly used by web servers to store data regarding to a user session. If an attacker is allowed to make a request after reading information stored in cookies, this could cause leakage of valuable information [13, 17, 19]. There are different ways to make a request to an external site, but here we present a policy that disallows changing the `location` property of the window to avoid such an attack.

Listing 15 shows how to construct such a policy by combining a listener (lines 1 to 4) and the predicate of the policy (lines 5 to 10). In the predicate, any attempt to change the `location` triggers the execution of `StateFnParam` that verifies if `cookieRead` is false. Otherwise, it is not allowed to change the `location`. Lines 10 to 13 install the policy.

Listing 15: Policy 6: Disable page redirects after `document.cookie` read in GUARDIA.

```
1  var lstnr = {
2         notify: (t,p,r,a) => {
3                 if(p === 'cookie'){
4                         setState('cookieRead',true) }}}
5  var noRedirect = Or(
6         And(Allow(['location']),
7                 StateFnParam(equals,'cookieRead',false)),
8         Not(Allow(['location']))
9  )
10 installPolicy({
11        whenWrite: [noRedirect],
12        readListeners: [lstnr]
13 }).on(window)
```

B.0.3 Policy 7: Allowing a whitelist cross-frame messages. Cross-origin communication using `window.postMessage` can lead to attacks such as Cross Site Scripting and Denial of Service. The policy below is intended to prevent these kinds of attacks by checking that

	GPL or DSL	Imperative or Declarative Specifications	Modified runtime enforcement?	Decoupled enforcement?
HV[9]	GPL	imperative	yes	no
ConScript[17][9]	GPL	imperative	yes	no
Richards et al. [21]	GPL	imperative	yes	no
Phung et al. [19]	GPL	imperative	no	no
JSand[1]	GPL	imperative	no	no
BrowserShield[20]	GPL	imperative	no	unknown
CoreScript [13, 29]	External DSL	declarative	no	yes but only policy code
Drossopoulou et al. [6]	External DSL	declarative	not applicable	not applicable
ObjectViews[16]	Internal DSL	partially declarative (inspired by AOP)	no	no
WebJail[25]	Internal DSL	imperative	yes	no
Guardia	Internal DSL	declarative	no	yes

Table 6: Overview of surveyed approaches with respect to the analysed design choices .

the origin URL of the message is white-listed. The predicate of the policy verifies, by means of `ParamInList`, that the second parameter of the invocation of `postMessage` is contained in a whitelist of URLs. If this is not the case, then the invocation of `postMessage` is denied.

Listing 16: Policy 7: Allowing a whitelist cross-frame messages in Guardia.

```
var urls = ['http://google.com', 'http://facebook.com'];

var whtList = Or(And(Allow(['postMessage']),
                ParamInList(1,urls)), Deny('postMessage'));

installPolicy({whenRead: [whtList]}).on(window);
```

B.0.4 Policy 8: Disallow string arguments to setInterval and setTimeout functions. This policy aims to disallow the execution of arbitrary code as described in [17]. Functions *setTimeout* and *setInterval* can accept a closure or string as callback argument. As such, these functions can be abused to run malicious code.

Listing 17 shows how we express a policy to restrict the execution of these functions to closures. In the policy below the execution of `setTimeout` and `setInterval` is permitted only if the first parameter of the invocation is a function.

Listing 17: Policy 8: Disallow *string* arguments to *setInterval* and *setTimeout* functions in Guardia.

```
var pol = Or(
    And(Allow(['setTimeout','setInterval']),
        ParamAt(typeOf, getVType(0,XFunction),Function)),
    Not(Allow(['setTimeout','setInterval'])));
installPolicy({whenRead: pol}).on(window);
```

B.0.5 Policy 9: Restrict `XMLHttpRequest` to secure connections and whitelist URLs. Phung et al. [19] prevent impersonation attacks using the *XMLHttpRequest* object by restricting its `open` method to whitelist URLs. Meyerovich et al. [17] propose a policy that enforces an HTTPS request when user and password arguments are supplied to the `open` method. Here we implement a security policy that compose these approaches.

Listing 18: Policy 9: Restrict `XMLHttpRequest` to secure connections and whitelist URLs in Guardia.

```
var startsWith = (a,b) => {return a.startsWith(b)}
var isHTTPS = StateFnParam(1,startsWith, 'HTTPS')
var pol = Or(And(
                Allow(['open']),
                ParamInList(1,urls),
                isHTTPS,
                Not(ParamAt(equals,3,undefined)),
                Not(ParamAt(equals,4,undefined))),
            And(
                Allow(['open']),
                ParamInList(1,urls),
                Not(isHTTPS)),Not(Allow(['open'])));
XMLHttpRequest = installPolicyCons(pol, XmlHttpRequest);
```

B.0.6 Policy 10: Only redirect to whitelisted URLs. Both Pungh et al. [19] and Meyerovich et al. [17], propose a policy to prevent redirection to another web site by means of changing the location property of the *window* and *document* objects.

Listing 19 illustrates this policy in Guardia. Redirections and setting of source locations are allowed only for URLs that are contained in a whitelist.

Listing 19: Policy 10: Only redirect to whitelisted URLs in Guardia.

```
const urls = ['http://google.com', 'http://facebook.com']
const whtList = Or(And(Allow(['location']),
                       ParamInList(0,urls)),
                   Deny('location'))
installPolicy({whenWrite:[whtList]}).on(document);
```

B.0.7 Policy 11: Disallow setting of `src` *property of images.* This policy was studied by [19] with the aim of preventing leakage of information by changing the source location of images, forms, frames, and iframes.

Listing 20: Policy 11: Disallow setting of `src` property of images in Guardia.

```
let image = document.createElement('img');
const pol = Or(And(
               Allow(['src']),
               ParamInList(0, url)),
               Not(Allow(['src'])));
image = installPolicy({whenRead:[pol]}).on(image);
```