Inlined Information Flow Monitoring for JavaScript

Andrey Chudnov
Stevens Institute of Technology
Hoboken, NJ 07030 USA
andrey@chudnov.com

David A. Naumann
Stevens Institute of Technology
Hoboken, NJ 07030 USA
naumann@cs.stevens.edu

ABSTRACT

Extant security mechanisms for web apps, notably the "same-origin policy", are not sufficient to achieve confidentiality and integrity goals for the many apps that manipulate sensitive information. The trend in web apps is "mashups" which integrate JavaScript code from multiple providers in ways that can undercut existing security mechanisms. Researchers are exploring dynamic information flow controls (IFC) for JavaScript, but there are many challenges to achieving strong IFC without excessive performance cost or impractical browser modifications. This paper presents an inlined IFC monitor for ECMAScript 5 with web support, using the no-sensitive-upgrade (NSU) technique, together with experimental evaluation using synthetic mashups and performance benchmarks. On this basis it should be possible to conduct experiments at scale to evaluate feasibility of both NSU and inlined monitoring.

1. INTRODUCTION

Many security issues are raised by web apps, especially so-called mashups that integrate in a single page code from multiple providers, including but not limited to third-party ads. Much of the utility of integrated apps derives from their manipulation of sensitive information including personal data and mission-critical data. Individuals and organizations have security requirements including confidentiality and integrity. Such information flow (IF) requirements pose challenges both for mathematical modeling and for usable policy specification techniques. There are also challenges in policy enforcement, i.e., information flow control (IFC). Enforcement is the topic of this paper. We assume that policy is given in the form of security labels attached to inputs and outputs, interpreted as specifying allowed dependencies, as formalized in the standard notion of termination-insensitive non-interference (TINI) [60].

Most client-side app code is written in JavaScript (JS), in part because of its flexible dynamic features —which exacerbate the difficulty of achieving security goals for mashups.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813684.

JS engines are highly engineered for performance, which has led some researchers to argue for inlined monitoring for IFC. The main contribution of this paper is an inlined IFC monitor that enforces non-interference for apps written in JS. We present the main design decisions and rationale, together with technical highlights and a survey of state of the art alternatives. The tool is evaluated using synthetic case studies and performance benchmarks.

On IFC. Browsers and other platforms currently provide isolation mechanisms and access controls. Pure isolation goes against integration of app code from multiple providers. Access controls can be more nuanced, but it is well-known that access control policies are safety properties whereas IF policies are hyperproperties [19], defined in terms of multiple executions. For example, TINI specifies information flow using pairs of executions. Prior work showed that even simple IF policies serve well to protect against common web attacks (e.g. [18]), so IFC may be useful without fully solving the problem of requirements specification. One approach to IFC is dynamic *taint tracking*, in which data is tagged with security labels that are propagated during execution. Taint tracking is very useful for catching bugs, misfeatures, and malware, but it does not enforce TINI because it fails to detect *implicit flows*: state changes that would have happened if a different branch had been taken. Indeed, information can also flow via covert channels such as timing and power consumption, but these are relatively difficult to exploit in the web setting and are outside the scope of this paper. In the literature, works on taint tracking often use the term "information flow", emphasizing the intended purpose rather than assured properties. In this paper, IF and IFC refer to TINI. Also, we focus on fine-grained IFC, as opposed to coarse grained policies which can be enforced by tagging at the level of processes [55].

One way to enforce TINI is by static analysis, using types, program dependence graphs, or theorem proving. Static analysis avoids the performance cost of runtime mechanisms and has the benefit of detecting insecurities prior to program execution. A complementary benefit of dynamic enforcement is that it can allow secure runs of a program even if it has other runs that are insecure. Static analyses can be excessively restrictive, due to conservative approximation of heap structure etc., and some require extensive programmer-supplied annotations. Features of JS, including `eval` (dynamically generated code), make static analysis particularly difficult. For this reason dynamic IFC is used in most work on IFC for JS, including this paper.

Goals and attack model for dynamic enforcement. The conventional approach to dynamic IFC builds on tag tracking, i.e., instrumenting the execution in such a way that potential flows can be "observed". Conceptually, the execution monitor performs an abstract interpretation of possible alternate runs, in order to soundly approximate TINI [16]. Another approach, secure multi-execution (SME) [20, 26, 46, 6], achieves non-interference using simultaneous concrete executions of the program, one for each security level, suitably orchestrated. In the next paragraphs the term "monitor" refers to any dynamic mechanism.

There are three primary correctness requirements for a monitored program under a given policy. First, *soundness* with respect to TINI: any violation of policy is detected by the monitor. The second requirement is *transparency*, which circumscribes allowed changes to behavior of the program to be monitored. Some changes are necessary, as the monitor must raise an alarm or halt execution when the program is poised to output a value at level incompatible with the level assigned by policy to that output channel. It may also halt execution because the mechanism is unable to ensure TINI even though there may be no actual violation. Such "mechanism failure" is inevitable for reasons of undecidability. Transparency has three aspects. If the monitor does not detect a violation on a given run, then (1) the outputs are the same as for the original program (a safety property) and (2) all outputs do occur (a liveness property). Finally, (3) if the monitor does detect a potential violation, this is reported. In that case, outputs may be suppressed or altered to avoid jeapardizing policy; the point is that the monitored program does not silently differ from the original's behavior. We call part (3) *frankness*. (To formalize TINI, it is convenient to model a monitor as diverging silently when policy might be violated, but this is seldom desirable in practice; cf. [31].) The third requirement is *permissiveness*: to minimize raising the alarm on executions that are semantically secure.

Formalization of TINI embodies an attack model. Such formalization is outside the scope of this paper but we sketch a model of the attacker derived from the *gadget attacker model* [1]. (AM0) The attacker does not have any special network privileges. The only messages that she can read are the ones directed at her own web server. (AM1) The attacker can introduce arbitrary JS code in the web-page. This might happen for one of the three reasons: (1) the user directed the browser to visit the attacker's web-site, (2) a legitimate third-party web-site included a frame or a script from the attacker's web-site or (3) the attacker has injected a script in a legitimate web-site via a cross-site scripting vulnerability. However, code injected by the attacker should comply to the standards of ECMAScript and HTML. We do not support vendor-specific extension. (AM2) The attacker cannot subvert or compromise the browser. We assume that the client machine running the browser has not been compromised and cannot be influenced by the attacker outside of injecting JS into pages. The JS engine implements correct semantics of standards-compliant ECMAScript and does not have vulnerabilities that allow to break the browser sandboxing model, or alter the semantics in any way. (AM3) The attacker knows the program (so they can learn from implicit flow), and they know the inliner, so their JS code may attempt to disable or subvert the monitor. (AM4) The attacker cannot observe power consumption and has a very limited view of timing as afforded by the API's available to JS programs.

In addition to the correctness requirements, deployability and performance are important considerations. Despite encouraging preliminary results, it is not obvious that SME scales well to policies with many security levels and to encompass APIs for web apps. In the rest of this paper, attention is focused on the conventional approach based on tag tracking, for which we henceforth use the term *monitor*.

Hybrid and NSU monitoring. Monitoring can achieve high precision, i.e., few false positives, by means of *flow sensitivity*, meaning that the security level of a storage location can change during execution. To achieve soundness, one approach is *hybrid monitoring* which depends on static analysis of possible heap locations updated in branches not taken. Based on such analysis, tags that represent an unbounded number of such locations may need to be updated at control flow join points [44, 51, 40]. It is quite difficult to implement sound static analysis for JS with acceptable precision and performance —especially for code that uses `eval`, which is ubiquitous on the web [50].

A technique that avoids the need for static analysis is *no sensitive upgrade* (NSU) [62, 4] which disallows raising the level of a location due to influences of branch conditions. In order to track these influences efficiently monitors can employ a stack of labels [51, 4], which is called a PC label stack. Each label corresponds to the label of a branch condition that has determined the path the program took to arrive at the current point. We use this technique, adapted to non-syntactic control-flow. The key point of NSU is that it achieves TINI without recourse to static analysis, at the cost of some restrictiveness (mechanism failures).[1] NSU can be even more restrictive than static analyses (indeed, must be [51]). Mitigation is possible in the form of explicit *upgrade annotations* that cause the monitor to preemptively raise the level on a location before a branch is taken.

In scenarios where legacy code is not a constraint, NSU-based monitoring with programmer-provided upgrades has been found viable in substantial case studies [25, 55] (albeit for coarse grained policies). Another possibility is to infer upgrade annotations through program testing [12]. However, less encouraging experience with an NSU-based JS monitor [28] led some researchers to devise a form of hybrid monitoring in which static analysis is used dynamically to infer upgrades; so correctness rests only on NSU but greater permissiveness can be achieved [27].

What is not known is the extent to which NSU (with upgrades) is compatible with existing code on the web. Several prior works have made advances towards building practical JS monitors, but considerable infrastructure is needed, e.g., to deal with many libraries, browser APIs, and obfuscated code used in practice. And the language itself is complicated and idiosyncratic. The JS interpreters in wide use are highly engineered, using in particular complex just-in-time (JIT) compilation of partial program traces. All JS IFC tools known to us omit at least one major language feature, and many lack support for DOM and other standard libraries. We report here on a new tool that should serve to evaluate NSU at scale.

[1]Because raising the level on a location, due to implicit flow, would be harmless if the location would have been written on all alternate control paths.

Inlining and goals of this project. To date, most implementations of IFC monitoring either modify a JS interpreter directly [11, 33, 26, 21, 59] or inject a custom interpreter with a plug-in [6, 28]. This may facilitate ensuring transparency and soundness, and it offers the potential for specialized performance optimizations. However, there is an ongoing "arms race" between browsers, with fast development of highly optimized engines, all based on JIT-compilation. At best, a modified JS engine is only of use with the associated browser, and it may be impossible to maintain without the assistance of the vendor. Some of the cited works on IFC either turn off JIT compilation (interpreter modification) or incur performance penalties by running JS code inside a JIT-compiled interpreter (custom interpreter plug-in). Work that does not harness JIT is doomed to have limited practical use. We also believe that it is impractical to modify the JIT compiler directly to generate monitored code. First, there are often several compilers working side-by-side, each geared towards different kinds of code. Second, the complexity of modern tracing JIT compilers is staggering.

Previous works suggest an alternative: inlining a monitor into the source program [17, 38]. In brief, the *inliner* turns a JS program into an *instrumented program.* The instrumented program runs in the context of additional JS programs: the *monitor core* and *API facades* explained in the sequel. Together these constitute a *monitored program.*

Inlining requires no browser modification. It can be deployed by an external HTML proxy, or at the server, or by a hook in the JS interpretation/compilation pipeline [36]. An obvious advantage of inlining is portability across JS engines. Just as importantly, inlined monitoring has the potential to benefit from JIT compilation. The engineering and maintenance challenge is proportional to the complexity of JS rather than the complexity of JIT compilers.

Nonetheless, the challenge is substantial. Unlike inlined reference monitors for access control policies, IF monitoring needs to be threaded through every execution step to track all flows. The interesting challenges include preserving the order of evaluation, which is important since expressions can have observable side-effects, and preserving the semantics of various kinds of references including the scope and prototype chains in JS. Inlined monitor code is vulnerable to attacks crafted to thwart the monitor, so the monitor operations, label storage, etc must be tamper-resistant.

The design of our inlined monitor was guided by the following five primary goals. (1) *Soundness and transparency.* (2) *Tamper-resistance.* (3) *Simplicity*: the inlining process and monitored program should be as simple as possible, to facilitate assurance of soundness, transparency, and tamper-resistance. Formal proof of correctness is not, however, in scope for our project. (4) *Modularity* is important as we would like to be able to modify aspects of monitoring quickly and easily for purposes of experimentation (e.g., support for different response to policy and NSU violations, and to explore variations on NSU like "permissive upgrade" [5, 10]). (5) *JIT-friendliness*: the transformed code should be amenable to efficient JIT compilation. An additional goal is for the monitored program to retain the structure of the original, which helps debugging the monitor, monitored program, and policy.

These design goals are in support of two main project goals. First, we are assessing the applicability and effectiveness NSU for web applications. Second, we are interested in practical performance of monitoring; our hypothesis is that it could be achieved by leveraging off-the-shelf JIT compilers, and we investigate whether it is true.

Contributions and outline. The *main contribution is an IFC monitor inliner for almost full ECMAScript 5, with support for web APIs. We report on experiments with performance benchmarks and also small but realistic mashups, for which we have built enough API and library infrastructure.* Our system is named JEST, for JS with Embedded Security Tracking.

Sec. 2 highlights IF issues in JS, only briefly since most have been described in prior work.

Sec. 3 describes the design and rationale for monitored programs and the inliner. The inliner handles actual scripts as found in web pages, handling all the complications of scripts embedded in HTML and providing support for tracking of IF across DOM operations. Although prior work has identified the main challenges of tracking information flows in JS, we have to address them in full, in all their guises. We argue why the design meets our goals.

Sec. 4 presents some experimental results, using an entirely unmodified JS engine. The inliner itself is written in Haskell and it's performance is not problemmatic; what we evaluate is performance of monitored programs. Experiments on benchmarks designed to evaluate JIT compilation exhibit on the order of 200× slowdowns depending on whether the inliner is configured to prioritize transparency. The results give some evidence that straightforward inlined monitoring with NSU can be competitive with the approach of modifying the JS interpreter, but do not show that our goal of JIT-friendliness has been achieved completely. We observe that the monitor core benefits nicely from JIT compilation but the instrumention we add to the monitored program prevented JIT compilation in our experiments using one JS engine.

The experiments also include case studies with mashups inspired by apps in the wild and which have interesting IF policies.

Sec. 5 provides further discussion of related work on IFC for JS. Sec. 6 wraps up with discussion of future prospects.

The JEST software distribution provides infrastructure for further experimentation and other investigations of IF in JS: wrappers for DOM API and ECMAScript standard libraries, and auxiliary open-source libraries and programs developed to support the inliner and the experiments. The source code of the inliner, supporting libraries, mashup case studies are released under an open-source license [15]. Detailed technical documentation is available in [14].

In the near term it is unlikely that any IFC monitor will have acceptable cost/benefit ratio for general client-side use. Even for taint tracking, prior works report quite significant slowdowns. Substantial slowdowns also result from code transformations to support "safe subsets" of JS. See Sec. 5 for some numbers. However, performance numbers including our own suggest that inlined monitoring can be practical for purposes of testing and security auditing (including forensics). In testing scenarios, significant performance degradation can be acceptable. In addition, reproducible tests can facilitate inference of upgrade annotations. IFC may also be practical for production use in situations where a security-sensitive mashup does not require a lot of client-side computation.

JEST supports all of ECMAScript 5.1 non-strict mode, except the `with` statement. This version is by far the most widely used for reasons of browser portability and performance. Support for `with` is possible if targeting a platform with an implementation of ECMAScript 6 Proxies.

Although this paper focuses on web apps, JS is used extensively outside of the browser context: many desktop and most mobile environments allow applications written in JS, which could benefit from IFC.

2. INFORMATION FLOWS IN JAVASCRIPT

Policies and example. In this paper we confine attention to policies in a simple, standard form. Policy has two parts. First, a fixed lattice of levels, where $l \sqsubseteq l'$ means information at level l is allowed to flow to l', which may be interpreted to mean "more secret" or "less integrity" or both. Second, fixed labels are assigned to input and output channels such as input forms on a page and network connections. The policy is interpreted to mean inputs at level l may influence outputs at level l' only if $l \sqsubseteq l'$. This *noninterference* property can be formalized in terms of two runs, where variation of inputs above l is allowed to cause variation only for outputs above l, for all l.

As an example, consider this third-party payment processor scenario. A web store employs a third party to process credit card payments. To avoid the need to deal directly with PCI compliance, it integrates payment processing at client side. At checkout an external `IFrame` from the payment processing provider is loaded which includes a form for the credit card information; the cost is provided via a `postMessage` from the main page to the frame. Upon completion of the credit card transaction, the payment processor returns a crypto-signed transaction summary (to prevent forgery by the user) which is sent to the merchant site via `XMLHttpRequest` to confirm payment. The merchant site should have no access to the credit card details and the payment processor should have no access to the order contents. The merchant site is allowed to disclose the total, the name and the zip-code part of the address (for credit card verification).

The merchant has incentives to impose a confidentiality policy given using three security levels: $\mathcal{M}$ (merchant-private), $\mathcal{P}$ (processor-private) and $\bot$ (public), such that $\bot \sqsubset \mathcal{M}$ and $\bot \sqsubset \mathcal{P}$; $\mathcal{M}$ and $\mathcal{P}$ are incomparable. In the payment IFrame we label the fields of the payment details form —credit card number, expiration month and year, CVV2— as $\mathcal{P}$ to prevent sending them to the merchant. We label the URL `<processor.domain>/pay` as $\bot$ so that the payment confirmation received from that URL or the error message can be sent to the merchant page. In the merchant checkout page we label all the fields and page elements as $\mathcal{M}$, except for the name and zip code form fields which are labeled $\bot$. The other channels in the merchant page are the payment confirmation `<mechant.domain>/payment` and order change submission `<merchant.domain/order_change` URL's, both labeled $\mathcal{M}$.

In our system policies are specified in a declarative language (though an investigation of usability of policy specification is not our goal). The system is modular enough to allow adding other ways of specifying policies with minimal changes: a policy is compiled to a JavaScript object with methods that give labels for locations and create new public labels. The policy writer can specify labeling of URI's, DOM elements and cookies.

Downgrading is needed for most practical requirements. In the example above, the individual purchase amounts are part of the order contents and considered secret, but their total must be revealed to the payment processor. Prior works suggest such policies can be specified by means of code annotations that designate some expression and program point where the value of that expression may be downgraded [53]. (Such annotations may be derived from higher level policies independent from the code; this important issue is beyond the scope of this paper.) The semantics of downgrading is subtle [53] but for our purposes adequate semantics is provided by prior work [7, 3, 58]. A monitor can implement downgrading by re-labeling, together with appropriate check for implicit flow. In our system downgrading policies can be specified in the application code using a call to a declassification function.

IFC challenges. In addition to heap locations and dynamic evaluation, there are other JS features that pose a challenge to precise tracking of information flows. Most of them have been studied in detail in [29, 28, 11]. We will remind the reader of the most interesting ones.

Variables and object fields are not the only storage channel: the structure of objects, arrays, DOM tree nodes and lexical environments [29, 52, 2] can store information too. We discuss a few examples to illustrate the point.

Arrays in JS are objects, with elements being fields with numeric names. Every array has a `length` field with interesting semantics: reading it will always give the index of the highest-numbered array element plus one, writing to it will cause all the elements that have indices higher or equal than the new value to be removed. Field names that are not valid indices do not interact with `length`. Consider this example:

```
var a = [1,2];
if (secret==1) a.length = 1;
output(low, "1" in a);
```

An array `a` is initialized with two elements. Depending on a secret its length is set to 1, which causes the second element to be deleted. By checking whether the element is present afterwards the attacker can leak one bit of information, which can be magnified.

A taint tracker could propagate labels in order to catch the indirect flow from `secret` to `a.length`. But, in order to account for the flow from setting `length` to the outcome of the `"1" in a` test, our monitor employs *structure* and *existence* labels on the objects and fields respectively [29]. This applies to arrays, objects and the DOM.

Local variables can be created and deleted too. Conditional creation cannot be done with static `var` declarations, because of *hoisting*. However, conditional creation of variables can be achieved using `eval`. For example, `if (secret) eval("var x;");` will create a new variable `x` if it didn't exist before. For this to be a flow channel it needs to be possible to detect existence as well. Unlike for fields, there are no built-in operators for enumerating or querying existence of local variables. However, one can use `ReferenceError` exceptions thrown when reading non-existent variables. Here is an example.

```
public = 1;
if (secret) eval("var x;");
try {x;} catch (e) {public = 0;}
output(low, public);
```

To prevent these attacks we currently disallow dynamic evaluation of code that introduces new variables in a high implicit context.

Another complication is the presence of unstructured control flow due to `break` and `continue` statements and exceptions. The presence of such flows means that syntax-driven analysis, monitoring and transformation rules found in the literature [38, 5, 29, 54] would miss or over-approximate implicit flows when applied to JS. There is also branching control-flow at the expression level. The fact whether a sub-expression is evaluated might depend on the result of evaluation another sub-expression. In addition to exceptions and the conditional expression (`-?-:-`), this occurs due to lazy evaluation of the logical operators `&&` (and) and `||` (or).

The ECMAScript standard does not define any input and output operations. Instead, it has to rely on the hosting environment —the browser in our case— to expose an API necessary for performing IO. Additionally, the browser exposes an interface for manipulating the web-page content, which serves as both IO channel and storage channel (structure of the document and values of document elements). Finally, there is the ECMAScript standard library. Some of these APIs can cause information flows across different parts of the API (e.g., assigning to the `length` property of an array can cause a `toString` method of another object be called), or even be invoked implicitly by the semantics. Implicit flows via exceptions, storage channels via structure and internal state, and side-effects (e.g., utility methods in the `Array` prototype object) are also an issue with the standard library. In order to maintain soundness information flows through the use of these API's need to be accounted for.

3. MONITOR DESIGN

We focus on the core components and design principles and discuss how they support our goals, as articulated in Sec. 1. Due to space limitations we cannot present an in-depth discussion of the design and implementation of the monitor and the inliner. More details could be found in [14, 15].

Soundness and transparency rely primarily on that of the information-flow semantics, which is similar to prior work [29, 52, 11], so we don't elaborate it. However, the semantics relies on the ability to mediate the operations of the underlying language semantics and the APIs, which is trivial to achieve in interpreter-based monitors and hard for inlined.

Performance depends on the scope of program instrumentation and whether JIT compilers generate efficient code for the monitor.

3.1 Additional challenges due to inlining

Inlined monitors live side-by-side with the monitored program. This makes them vulnerable to attacks from the program that may try to tamper with the monitor state or implementation. Protection is not easy since the monitored program needs access to the monitor. Another issue we need to consider is discrepancies between browser implementations that might expose differing APIs. Fortunately, addressing these challenges is possible with semantic mediation and API emulation.

As noted in [28, 24, 13], the semantics of ECMAScript and DOM is full of edge-cases and implicit mutual dependencies: for example, applying the `+` operator may cause a call to a `toString` function defined in the standard library — or even to a user-defined one! The monitor must track all these interactions.

Finally, being implemented in JS itself, the monitor should play by its rules and neither has access nor can modify the inner workings of the interpreter and the libraries, such as the internal algorithms and state, as defined in the specification. This complicates achieving mediation while preserving transparency.

3.2 Principal design choices

The goals and challenges have guided us to make the following design decisions.

Boxes. Association of security labels with values plays an important role in the monitor design. Previous work has used *shadow variables and fields* [54, 17], *sparse labeling* [11, 4], and boxing [29, 11].[2] We argue that boxing is more appropriate for inlined monitors in JS. Our boxes are objects with four fields: `v` for the value, `l` for the security label, `t` for the type tag and `m` for meta-data.

The primary purpose of boxes is to store values together with labels. This simplifies storage, eliminating the need to distinguish between fields and variables as with shadow locations. It also simplifies reasoning about access to the labels. It also allows us to adopt an *important invariant: all the transformed (sub-)expressions evaluate to boxes.* Finally, it allows us to store extra meta-data used in precise modeling of the internal algorithms of both the HTML and ECMAScript specifications in an efficient way.

On the face of it boxes are a terrible idea because they require an extra level of indirection to all data accesses and more storage. Yet, we have observed that they simplify both the monitor core and the inlining algorithm and, at the same time, allow for more efficient execution in the JIT.

JavaScript's dynamism has led modern JIT compilers to adopt dynamic —instead of static— analysis for optimizing code generation. The most common approach,[3] which applies to V8 and SpiderMonkey, is dubbed *tracing method JIT compilation*. Compilers have two tiers. The first generates native code for function bodies on their first invocation and supports all JS features at the price of suboptimal code. It also inlines a profiler, which collects information about function invocations: their number and the types of the arguments and return values. If a function is called often enough (implementation specific) and with *stable types* (the types of arguments are the same across invocations), the second-tier compiler is invoked. It generates optimized code for a subset of the language and arguments of specific types (this is called *run-time type specialization* [23]). Op-

[2]The term boxing comes from implementations of dynamic and functional languages where boxed values are those that contain additional meta-data.

[3]This is a simplified account that corresponds to state-of-the art at the time of writing. JIT compilers are constantly evolving and precise documentation regarding their architecture is scarce and often outdated. Our experiments, however, have confirmed the facts described here.

timized code can be executed up to 20x faster [23] than the unoptimized.

Inline caches [30] allow translating field accesses to memory lookups using constant offsets. This requires dynamically inferring *hidden classes*, which are akin to structural types for objects that account for the order of field initialization. Note that hidden classes are distinct types for the purposes of specialization. Enforcing the box invariant allows the monitor core and API facades to be compiled into type-specialized code very early in the execution, which we have verified experimentally. It is also faster than sparse labeling. Recall that sparse labeling mandates labels to be attached to values if the former are non-bottom (e.g., not "public" for the classic confidentiality policy). But that means that the monitor is going to operate on many different types of values: all the JS run-time types in addition to the hidden class of boxes. This prevents monitoring code to be considered type-stable and prohibits optimization. Moreover, it requires additional run-time overhead to determine whether a given value is boxed or unboxed.

Functional monitor core. The monitor needs to perform operations like comparing (`leq`) and joining (`join`,`join2`) levels, tracking implicit flows (`enter`, `exit`, `update` and `push`), applying the NSU and structural restrictions (`nsuCheck` and `nsuxCheck`). These operations are part of the monitor core. They are invoked often, and we'd like them to be fast.

Type stability is a necessary, but not sufficient condition for optimization: the code should be confined to a single function, with many compilers imposing restrictions on the size of its body and requiring it to be declared in and accessed via a variable (as opposed to an object field). We design the monitor accordingly: as a collection of small functions stored in variables. This is good software engineering practice and it simplifies static type checking and testing too. More importantly, it helps the JIT generate optimized code: The functions assume arguments and return values only of a limited number of types (mostly, boxes) and avoid costly language features like `with` or exceptions that invalidate optimizations. Small size and limited functionality causes the functions to be used often in the instrumented program and force early specialization, which we have verified by investigating the tracing logs of JIT compilers.

Keeping the monitor operations as functions also simplifies inlining, e.g., helping preserve the order of expression evaluation. To give a specific example, consider the expression `e+x`. The naïve way to rewrite it, assuming the box invariant (and ignoring `t` and `m` fields), would be as

```
{ l: e.l ⊔ x.l, v: e.v + x.v }
```

However, what if `e` has side-effects, e.g., it is `y++`? It would be evaluated twice and will cause `y` to be incremented twice, yielding an incorrect result. Passing expression arguments to functions is an easy way to force evaluation of sub-expressions: in this case, the sum would be rewritten as `opadd(e',x)`, where `e'` is the transformation of `e`.

Operation emulation. ECMAScript semantics is complex and relies on internal algorithms that are not visible to the user and cannot be altered or mediated. Yet, they involve complex information flows, which must be modeled. For fine-grained tracking the monitored program must *emulate* parts of the semantics explicitly. The emulation models internal behavior and state, allowing more precise reasoning about control- and information flows.

```
function opadd (l, r) {
 'use strict';
 var pl = ToPrimitiveBox(l),
     pr = ToPrimitiveBox(r);
 return primbox(pl.v+pr.v,pl.l.join(pr.l))
}
```

Listing 1: `opadd` monitor operation

```
function ToPrimitiveBox (b, hint) {
  'use strict';
  if (IsPrimBox(b)) return b;
  else return DefaultValue(b,hint);
}
```

Listing 2: `ToPrimitiveBox` monitor operation

Using addition as an example, we note that, like many other operations, it involves dynamic type coercion. If the type of the first argument is coercible to a string, the operation acts as concatenation. Otherwise, it works like ordinary number addition. Now, consider addition of a string to an object: `""+{}`. It would cause the object to be converted to a string, which is the result of calling a `toString` method of the object, normally inherited from the `Object` prototype. However, if the user redefines the method, that would be called instead. There is an issue with this: the user-defined method is going to be transformed by the inliner and will operate on boxes. The run-time, which doesn't know anything about boxes, is going to call the function, providing an unboxed value for `this` and treating the return value as unboxed (not boxed) string.

To address this we perform explicit type conversions of boxed values and emulate certain aspects of JavaScript semantics and library behavior. Addition is rewritten into a call to the monitor operation `opadd` (listing 1), which takes two boxed values and calls `ToPrimitiveBox` in order to capture the possible side effects due to implicit type coercion. `ToPrimitiveBox` (listing 2) checks whether the box stores a primitive value (using the `t` flag field) and calls `DefaultValue` if its not.

Readers familiar with the ECMAScript specification will recognize `DefaultValue` as the name of an internal operation; this is modeled by its counterpart in the monitor RTS (runtime system), including calling `toString`. We end up reimplementing certain aspects semantics of ECMAScript, similar to [28], but unlike them we avoid reimplementing everything and instead rely the underlying semantics as much as possible. For instance, `opadd` does not reimplement the addition operation in that it does not query and convert argument types in case they are primitive: it defers to the actual language run-time by using the native `+` operator. This is safe because primitive value conversions are side-effect free.

A few additional examples of monitor operations are:

- `primlow(x)` takes a primitive value and boxes it with the public label,

- `assignField(rhs,o,f)` performs both NSU and structural checks and provide support for the special semantics of arrays.
- the three functions `invokeFunction(f,...)`, `invokeMethod (o,f,...)` and `newObject(c,...)` are used to perform calls; they account for the implicit flow and provide the correct box to be bound to the special parameter `this`. The latter is not possible with native function calls.

API facades. Like language semantics, APIs need to be mediated too: They often involve complex control flows and information flows, and they need to be instrumented to work with boxes.

API's are exposed as JS objects, functions and computed fields. Their mediation in inlined access control monitoring has been studied in [42, 37], but also see [35, 45, 22] for investigation of code isolation. It was found that simply installing wrappers by redefining object fields is prone to attacks, like prototype poisoning: the attacker can redefine a function that is often called implicitly, e.g., `Object.prototype.toString` from our previous example, to subvert the monitor. Another complication is the fact that some API components cannot be redefined (read-only fields), or meaningfully wrapped (like the `length` field of arrays).

Once again, we opt for either partial or complete emulation of APIs which is implemented in *API facades.*[4] They are ordinary JS objects, functions and values that implement box- and information-flow-aware interfaces that mirror the native ones. Some of the facades rely on the native APIs for performance, simplicity or IO (e.g., the DOM API). The common responsibilities of facades are to unbox arguments for native APIs, box return values with levels that capture information flows, account for implicit flows due to exceptions that might be thrown, apply restrictions to the observable memory side effects, and apply the security policy at the IO endpoints. Facades never pass boxed values to native code, which is unaware of box semantics. They unbox the values in a limited manner: For objects, most API's either do not inspect the object graph (the only exception we have found so far is `window.postMessage`) or coerce objects to primitive values anyway. In the latter case we coerce object boxes to primitives beforehand.

This approach also allows us to avoid dealing with the differences in browser APIs, which can be significant, and to mask (for soundness) the APIs that we don't support yet. Facades are written in pure JS and rely on the monitor core, property descriptors and box meta-data to maintain transparency and enforce mediation.

Script consolidation. In web pages scripts can exist in multiple places: `<script>` tags, event handlers and URL-properties. This makes it impossible to wrap the whole program in a closure, which we want to do for confinement (Sec. 3.4). To address this problem we have developed *script consolidation* which extracts all the statically known pieces of JS in the page and puts them in a single inline script, while preserving the behavior of pages that follow the HTML 5 standard. All occurrences of scripts, including event-handler properties (e.g., onclick) of HTML elements are subject to consolidation. Dynamic scripts, however, like those to which `eval` is applied, are handled separately.

In a few edge cases that involve with deprecated HTML features, consolidation is not transparent. These features are deprecated because they enable execution of scripts to affect the state of the HTML parser. Consolidation is not transparent for such effects.

Script consolidation simplifies handling of a few things, such as event handlers. It allows to store the RTS in function statements instead of properties of the global objects, which is nice for the JIT. However, it is not a hard requirement for making the monitor work. If consolidation was not used, we could have stored the monitor core and state in global variables. But those are in the global object and the `for..in` statement allows enumerating its properties, which would make them vulnerable to tampering. We could have instrumented the statement to skip over our prefixed variables, or used property descriptors to make the properties non-enumerable. Both solutions incur extra run-time overhead; the latter because, at least in some JITs, access to fields with non-default descriptors is not optimized.

3.3 Principles of implementation

Our goal is to minimize overhead for well-designed and optimized programs, while making it possible to run the rest and maintaining transparency and soundness. Often these goals are in conflict. The implementation follows a few general guidelines to improve performance and aid mediation.

Leveraging fast operations. Type specializing compilers support only a subset of JS, so leveraging them requires restricting the vocabulary used when implementing the core and facades. For example, 31-bit integers are known to be stored unboxed in V8, so we use those as much as possible, e.g., for the box type tag and the bit-vector label representation.

We also leverage native variable lookups and scope chains, as well as native field lookup and prototype chains. An alternative is explicit emulation, which requires costly enumeration and recursion.

Laziness. Fast startup is deemed very important for web applications and is one of the reasons for using dynamic optimizations. But it can be challenging for an inlined monitor that supports a lot of API's. To aid that, we initialize all the API components lazily. For example, the `Object` facade constructor is not initialized until its first use. And even then, its prototype fields remain unevaluated. We use ECMAScript accessors for that.

3.4 Structure of a monitored program

The logical structure of the monitor at run-time is shown in figure 1. The arrows show flows of control and information. DOM API and ECMAScript standard library are part of the browser (shaded). The instrumented program is the only component that depends on the original. The run-time representation of the security policy is generated from a declarative specification, introduced in section 2.

The syntactic structure of a monitored program is shown in listing 3. It makes heavy use of anonymous function closures and local variables. We need to ensure access to the monitor core and the facades from the instrumented program while ensuring they are tamper-proof at a reasonable

[4]The term is chosen to avoid confusion with ECMAScript 6 proxies.

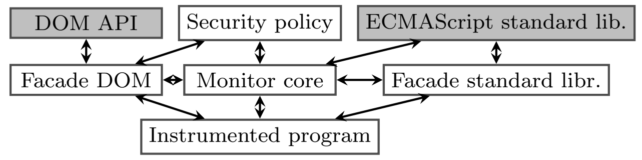

Figure 1: Structure of a monitored program (data flows)

```
(function () {
  function xxstop (...) {..}
  /* monitor declarations */;
  var xxpolicy = /*...*/;
  var xxglobal = new (function () {
    /* global facade constructor */
  })();
  return (function () {
   var /* API lockout variables */;
   with (this.v) {/*instrumented prog.*/};
  }).call(xxglobal);
})();
```

Listing 3: Structure of a transformed program (abbreviated)

cost. This is done by prepending a random prefix to the name of every monitor variable; here and in the rest of the text this prefix is `xx`. We make sure that the prefix does not clash with the existing program variable names. Because we mediate all the operations and expose our versions of APIs, this leaves brute-force guessing as the only option for the attacker. The only way to implement guessing variable names at run-time is using `eval`. The prefix can be thought of as a secret shared between the monitor core and the instrumentation. Guessing that secret gives the attacker nothing, because the only way to refer to it is inside `eval`, and the inlining procedure for dynamically evaluated code outlaws any references to the monitor variable.

For transparency reasons we need to support aliasing of the properties of the global object facade with global variables. We use the `with` construct for this. However, most JITs don't optimize function bodies that contain `with`. We have an optional optimization for programs that don't add properties to the global object and refer to them as variables. This optimization avoids the use of `with` and gains a significant increase in performance due to JIT optimization (section 4.2).

The lockout variables (Listing 3) are locals with the same names as those of top level API's: `Object`, `Array`, `window` etc. They allow to retain transparency in case the fields of the global facade exposed via `with`, are deleted: Without the lockout variables, the original API's would have been exposed. Note that the latter would not jeopardize security and isolation, because all the means of accessing the API's —field access and function calls— are mediated by the monitor run-time. Mediated operations invoked on native API's would fail with an exception due to the violation of the boxing invariant. While unpleasant, this is not a security vulnerability. This also takes care of the cases when the surface of the browser API is larger than expected by the inliner.

Finally, this approach allows separating the monitor from the global scope, which simplifies testing and benchmarking.

3.5 Inlining

The architecture of the inliner is presented in figure 2. The original program is either a standalone JS program or it is in a web page. If it is in a web page it first needs to go through script consolidation to produce a single JS program equivalent to the multiple scripts (Sec. 3.2).

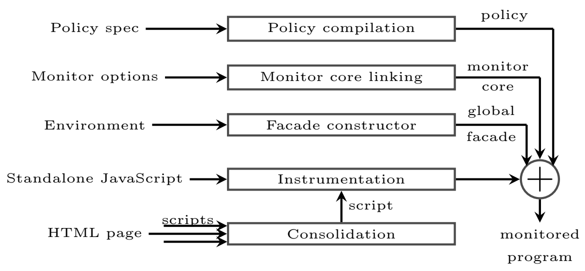

Figure 2: Inlining algorithm

It is useful to think of the inliner in terms of the traditional compiler, which has a run-time system (monitor core and API facades), a front-end, conversion to intermediate languages (desugaring), a static analysis pass and code-generation.

Desugaring. We simplify the program slightly to aid establishing the box invariant. In particular, rewrite function statements into function expressions and decouple variable declarations from their initialization. Note that both transformations follow the hoisting procedure dictated by the ECMAScript specification.

We have chosen not to do more aggressive desugaring in an attempt to keep the structure of the instrumented program similar to the original. Moreover it is challenging to correctly desugar to a small core (see Sec. 5).

Static analysis. We perform control flow and exception analysis to determine control dependence regions of branch points. We use the definition of control dependence regions due to Barthe et al. [8]. The region inference algorithm works with the intraprocedural control-flow graphs and is build on top of [43]. Graphs are at expression granularity (see Sect. 2 for motivation). For each region we identify the guards, the entry point and exit points. Those are used to guide the instrumentation to insert operations that manage PCLS (the PC label set): `enter` at the entry, `exit` at the exits and `update`/`push` at the guards.

These operations allow tracking at the expression level by interleaving stack operations with subexpression evaluation: e.g. `update` takes and returns a box, in addition to updating the guard label, and `exit` can return the argument box in addition to discarding the PCLS record for the region.

Our control-flow analysis is intra-procedural because in the general case it is impossible to construct a precise call graph for JS programs. Instead, we adopt a dynamic approximation. We know that the control-dependence region of the exception source extends to the end of the innermost `catch`/`finally` clause. That point is a conservative approximation of the merge point of any exception source within the try block. Hence, we can remove the PCLS records representing the corresponding implicit flows. The complication is that we don't know how many elements we need to remove, as the corresponding branch points can be in another function. The following observation helps: The try statement contains all the implicit flows due to exceptions, so the aggregate level of the PCLS should be the same before and after it. This invariant allows us to put an easily enforceable and sound approximation on the control-dependence regions of inter-procedural exceptions. To this end, in addition to the operations for adding, removing and updating PCLS records we have introduced two new ones:`remember(id)` and `restore(id)`, where `id` is a numeric identifier which is syntactically unique for every try-catch statement. The operations allow to save and restore the state of PCLS. Using these two operations we can transform a `try-catch-finally` statement as shown in the listing below.

```
try {xxremember(<id>); /* try body */}
catch (x) {/* catch body */}
finally{xxrestore(<id>); /*finally body*/}
```

Program instrumentation. Fig. 3 shows the instrumentation algorithm. We show a few transformations to give a flavor of what an instrumented program looks like. The code in Listing 4 contains a while statement with a variable assignment, infix expressions, and a conditional expression. The transformed version (Listing 5) deals with control flow within expressions and assignments, using the `push` and `pop` monitor functions which both pass values and have effects on the level stack. Listing 6 shows function declarations and calls, transformed to Listing 7.

The rewriting rules, defined as a syntax- and annotation-directed translation, are presented in tables 1 and 2. The rules should be read as "if an AST node matches the production and has, at least, the annotation, then replace it with output". Italic denotes arbitrary sub-statements or expressions. The rules assume rewriting is done in a bottom-up fashion, so the sub-expressions/statements are already rewritten.

Accessor properties. *Getters* and *setters* are supported by the monitor. In order to enforce the boxing invariant we do not rely on the native accessors. We emulate them explicitly, so we can bind `this` in getter and setter functions to the object box instead of the object as would have been done by the native semantics. Refboxes are objects with fields `"t"` with an appropriate flag, `"g"` and `"s"` that store the getter and setter functions respectively. These functions are invoked in `readField` and `writeField` monitor operations. Refboxes are stored in properties that would have had accessors defined. The facade for `Object.defineProperty` takes care of the conversion between property descriptors and refboxes.

Eval. The `eval` function allows interpreting a string as a JS program. Precise tracking of information flows in dynamic code requires performing inlining on the code before evaluating it [38]. Previous work implemented the inliner in JS [28, 54], which can be used in `eval` as well. Our inliner is implemented in Haskell. We support inlining of `eval`'ed code with the help of an inlining HTTP proxy server. This suffices for two of our deployment scenarios: browser and proxy server. To support the server-side deployment scenario, the inliner needs to be added to the monitor core. We could achieve that using a compiler from Haskell to JS [56].

Declassification and upgrade. Another addition to the environment is a function `declassify(e,c)`, which is allowed in source programs to express policy. In the monitor, it downgrades the label on `e` to that of the channel `c` while enforcing robust declassification [41]: it is a violation if the PC level is higher than the initial label on `e`. The explicit label upgrade operation is also exposed in the API.

4. EXPERIMENTS

We are interested in practical yet sound information flow enforcement. The two longstanding questions for NSU-based monitoring are whether it can achieve adequate permissiveness and performance without sacrificing soundness or transparency. We believe these questions don't have a satisfactory answer in the literature. We don't claim to have the ultimate answer, but instead offer additional evidence that this approach is moving towards practicality.

For permissiveness assessment to be conclusive one needs to study existing applications and provide comprehensive policies that account for all the legitimate flows. This is a daunting task, as modern web applications are very large, often using multiple JS libraries, a wide variety of browser API's, and they are often obfuscated. That's why we have opted to create our own mashups to serve as case studies, but inspired by the mashups we have seen "in the wild". The mashups we have developed are only mock-ups in a sense that they include only the bare minimum to demonstrate the patterns of mashup component interaction (using the currently recommended APIs) and create possibilities for both legal and illegal IF.

We have covered a variety of different programming idioms when implementing the mashups. For example, attaching event handlers by specifying the corresponding attributes of HTML tags versus using `addEventListener` in the JS program itself; different ways of mashing up content: using IFrames or script tags; using inline scripts versus external scripts (e.g., using the "src" attribute). Every example application has one security policy and multiple versions of components, some of which conform to policy, while others do not.

4.1 Securing mashup applications

We focus on web applications with interesting IF policies. Most are mashups combining two or more JS programs from different service providers on one page. From an IF perspective mashups are the most interesting: the providers are often mutually distrusting or have a legal obligation not to disclose information to a third party —yet certain flows should be allowed in order for the application to be of use.

Third-party payment processor. This was discussed in Sec. 1. A malicious version of the merchant page sends order details together with the final price to the processor. The malicious processor sends payment information to the merchant.

Advertisements. An Internet radio service, similar to Grooveshark, Last.fm or Pandora, hosts ads from a ticket vendor, e.g., Ticketmaster, that list upcoming shows. Similar to how most ads are hosted nowadays, the service includes a third party script. The script crawls the page and creates a new script tag with a URL that contains keywords.

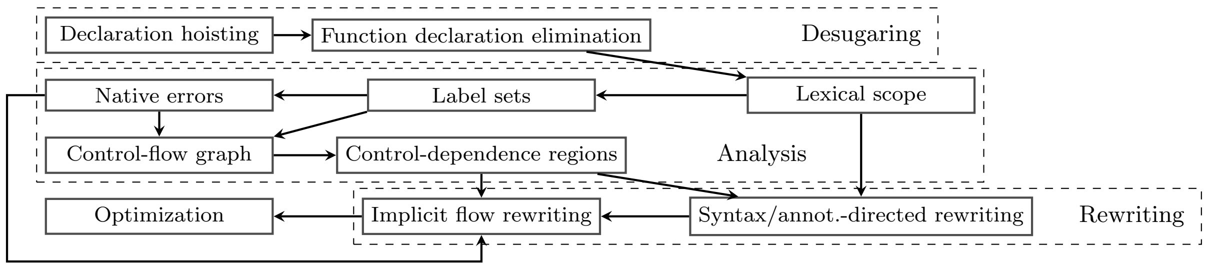

Figure 3: Instrumentation algorithm

Production	Annotation	Output
"$abcde$"		xxprimlow("$abcde$")
function f ($x,\ldots,z$) {ss}		xxfunlow(function f ($x,\ldots,z$) {ss})
x	OuterFunRef	xxfunlow(x)
x	$\emptyset$	x
o.f		xxreadField(o, xxprimlow("f"))
o[f]		xxreadField(o, f)
o[f]($e_1,\ldots,e_n$)		xxinvokeMethod(o,f,[$e_1,\ldots,e_n$]))
f($e_1,\ldots,e_n$)		xxinvokeFunction(f,[$e_1,\ldots,e_n$])
new C($e_1,\ldots,e_n$)		xxnewObject(C,[$e_1,\ldots,e_n$])
void e		void e
!e		xxoplnot(e)
e_1||e_2		(function (x){return xxToBooleanBox(x).v ? x : xxjoin2(e2, x.l);})(e_1)
e_1&&e_2		(function (x){return xxToBooleanBox(x).v ? xxjoin2(e2, x.l) : x;})(e_1)
e_1+e_2		xxopadd(e_1, e_2)
x++		xxvarpostfixinc(x)
x = e	NotDeclared	xxassignVar((function () {try {return x} catch (ex) {return xxglobal.x=xxinitVar()}})) e)
x = e	$\emptyset$	xxassignVar(x, e)
o[f] = e		xxassignField(x, f, e)
$e_1,\ldots,e_n$		$e_1,\ldots,e_n$

Table 1: Selected rewriting rules for expressions

Production	Annotation	Output
if (g) t else e	GuardIndex(n)	if (xxToBooleanBox(xxupdate(n, g)).v) t else e
while (g) ss	GuardIndex(n)	while (xxToBooleanBox(xxupdate(n, g)).v) ss
for (i; t; p) ss	GuardIndex(n)	for (i; (xxupdate(n, t)).v; p) ss
for (x in o) ss	GuardIndex(n)	for (xxtemp = (xxupdate(n, o)).v) {x = xxprimlow(xxtemp); ss}
var $x,\ldots,z$		var x=xxinitVar(),$\ldots$,z=xxinitVar()
with (o) ss		with (xxadaptForWith(o)) ss
s	CDREntry(n)	xxenter(n); s

Table 2: Selected rewriting rules for statements

Suite	Benchmark	Mean run-time of the original	Mean run-time of the monitored	Optimized	Slowdown
SunSpider 1.0.2	Access Binary Trees	0.0027s	0.9267s	No	342×
SunSpider 1.0.2	Access Fannkuch	0.0532s	6.143s	No	116×
SunSpider 1.0.2	Bitops 3 Bit Bits in Byte	0.0043s	1.355s	No	315×
SunSpider 1.0.2	Math partial sums	0.0077s	0.7866s	No	101×
Kraken 1.1	JSON Parse Financial	0.0645s	0.5132s	No	8×
SunSpider 1.0.2	Access Binary Trees	0.0025s	0.5949s	Yes	231×
Kraken 1.1	JSON Parse Financial	0.0661s	0.5057s	Yes	7.6×

Table 3: Selected benchmark results

```
while (a > 10) {a -= b * 0.1; b = b > 10 ? b : --b;}
```

Listing 4: Original

```
xxenter(1);
while (xxToBooleanBox(xxupdate(0, xxopgt(a, xxprimlow(10)))).v) {
  xxassignVarOp(xxopsub, a, xxopmul(b,xxprimlow(0.1)));
  xxassignVar(b, xxexit(xxToBooleanBox(xxpush(xxopgt(b, xxprimlow(10)))).v ? b :
      xxprefixdec(b)));
}
xxexit();
```

Listing 5: Transformed from Listing 4

```
function f (x){x(1);}; f(function g (y){alert(y); g(y);});
```

Listing 6: Original

```
var f = xxinitVar();
xxassignVar(f, xxfunlow(function (x) {xxinvokeFunction(x, xxprimlow(1));}));
xxinvokeFunction(f, xxfunlow(function g (y){xxinvokeFunction(alert, y);
                                              xxinvokeFunction(xxfunlow(g), y);}));
```

Listing 7: Transformed from Listing 6

This second script contains a declaration of a variable containing the list of upcoming shows, which is read by the crawler script, rendered as HTML and included in the web page. The policy is that, for better targeting, the crawler script is allowed to read the user's playlist and history, as well the Zip-code. However, for privacy and security reasons, the crawlers should not read the user's name, authentication credentials or any other sensitive information.

Such a policy might be provided by the radio service in hopes to distingish itself as more privacy-aware than others. Perhaps more likely is that a hardened browser designed to meet requirements of some government agency may have default policies concerning authentication credentials. It is not our aim in this work to solve the problem of motivating organizations to specify policies.

Third-party authentication service. A 3rd-party authentication provider gives the ability to integrate a login form via an IFrame. It verifies the authentication credentials with an asynchronous request to the auth server. The server returns an authentication token, which is, in turn, communicated back to the hosting page via postMessage. A web-site that would like to authenticate a user is not allowed to see the authentication credentials.

Currency converter. This web-based currency converter is not a mashup, but still has an IF policy. The conversion is done on the client side. But, in an effort to be timely, it pulls the conversion rates from the server each time. The policy is that neither the original, nor the resulting amounts are disclosed to the server — and the malicious version does just that.

Experiment setup. We accessed the mashups —both the benign and malicious versions— in the browser through an inlining proxy. The proxy inlined the monitor that enforced the intended policies for the applications. We then interacted with the applications and observed their behavior and whether there were any security violations.

Findings. The monitor could run the benign versions of case-studies with one declassification (payment) and upgrade (ads) annotation each. The attacks from malicious versions were all successfully stopped. This is consistent with earlier findings in [28]. We have not found any new kinds of vulnerabilities.

4.2 Performance benchmarks

Synthetic performance benchmarks are the common way of measuring performance of JS run-times. At the time of writing SunSpider, Octane and Kraken were the most widely recognized and used. Yet, such benchmarks are also contentious [48] because they focus on numerical computations and algorithms on data-structures. Automatic construction of benchmarks from widely used websites has been proposed as an alternative [49], but we opted out of using it due to heavy use of `eval` for implementation which complicates reliable performance measurements in our implementation. Also, use of `eval` disables optimization in all the JIT compilers we've studied.

We still wanted to get a sense of performance overheads, so we've chosen SunSpider and Kraken, which could be run stand-alone easily. In addition, we use them as an extra test of transparency, since they check correctness of results. We recognize that these benchmarks are not representative of web apps —particularly the security critical ones— and constitute the worst-case scenario for the monitor.

We select a subset of SunSpider and Kraken in an attempt to maximize diversity, but minimize the additional API support required. SunSpider tests are heavy on mathematical computations, bitwise operations and string processing, while most of Kraken is about signal processing and cryptography, along with an implementation of A^* and two JSON-related benchmarks. We perform two kinds of measurements. First, we use the Benchmark.js library to measure performance accurately, factoring out interpreter and RTS startup times and fluctuations in the execution time due to garbage collection, JIT compilation etc. We compare the running time of benchmark code instrumented with the monitor following a trivial policy to the uninstrumented one. We observed $101-364\times$ slowdown in the mean running time depending on the benchmark and the inlining optimization. We have a mode that trades off some transparency for up to 40% speedup by omitting the `with` statement in the monitor structure (Sec. 3.4). See table 3, as well as [15, 14] for more details.

We also compare our performance to the closest related work, JSFlow [28], which is discussed in section 5. It was impossible to use the benchmarking library with JSFlow without invasive modifications. Instead we used the Unix `time` command to measure one run of the benchmark without instrumentation, with our monitor and with JSFlow. This approach does not account for measurements fluctuations (observed to be about 1%) and includes the interpreter startup time. In this test the inlined monitor exhibits a $15.6\times$ slowdown and JSFlow is $1680\times$ slower compared to the original. Detailed data as well as instructions on how to reproduce the experiments is available in [15, 14]. The numbers for JSFlow are consistent with the authors' observations reported in [28] and private communication.

All measurements were done in Node.js v0.10.25 with all the performance optimizations enabled.

While the performance results look uninspiring, let us put them in perspective. First, the results are from running synthetic, computationally intensive benchmarks on a state-of-the-art JIT compiler that was designed to run these particular benchmarks fast. Second, the performance analyses in closely related work either report overheads for hard-to-reproduce "macro benchmarks" or compare performance slowdown against interpreters that don't use JIT.

Due to reactive nature of our case-study applications reliable measurement of performance requires additional infrastructure. One promising approach would be to adapt them, as well as examples from previous work, for performance testing by removing reactivity.

5. RELATED WORK

Fragoso-Santos and Rezk implement an inlined NSU-based monitor for a subset of ECMAScript 3rd edition and a small but challenging subset of the DOM API [54, 2] including "live collections". No data on performance is provided. The JS subset is very small, omitting non-syntactic control-flow, exceptions, the `with` and `for-in` statements, the `in` and `new` operators, as well as flow via the standard library and implicit type conversions. Thus there are relatively few possibilities for an attack on the monitor, and fewer peculiarities of the IF semantics. This allows for inlining to use "shadow" variables and fields to store labels, which is relatively simple and should cater for performance. The authors provide formal proofs of soundness and transparency.

The complexities of JS and of IF strongly motivate formal verification for assurance of monitors. Several lines of work on JS IFC monitoring provide proofs of soundness (and transparency in some cases). However, these proofs are all for formalizations that idealize (in varying degrees) from the implemented systems. We are aware of no verified implementation of an information flow monitor for JS. To make verification more tractable, it is attractive to minimize the complexities of JS by distilling to a small core. This is difficult to achieve for full JS, as discussed in detail by [24].

Just et al [33] have added an IF monitor to the WebKit JS interpreter. The monitor appears to use the NSU approach and includes full JS support including `eval`. The authors appear to be the first to recognize the importance of using control-flow graphs for accurate and sound tracking of implicit flows due to unstructured control flow (break, continue, return, exceptions). Implicit flows due to element existence are also discussed. Although exceptions are discussed, the static analysis does not deal with exceptions; in fact the authors question whether the approach using control-flow graphs will work with exceptions. Performance tests using synthetic benchmarks have been performed reporting $2\times$ to $3\times$ slowdown compared with a non-JIT interpreter. A qualitative experiment with a short JS program that didn't use the browser API has been performed as well.

Building on [33], Bichhawat et al [11] modify the JS bytecode interpreter to track IF, handling implicit flows using immediate post-dominator analysis of intra-procedural control flow graphs built on-the-fly. Semantics of the bytecode is formalized and used to give a formal proof of soundness. The monitor implements permissive upgrade [5], including its non-trivial extension to arbitrary security lattices [10]. It is also the only one to implement sparse labeling [4]. With sparse labeling, they report 0 to 125% run-time overhead in synthetic benchmarks (SunSpider) with the average being 45%, and 7 to 42% overheads in macro benchmarks (web sites), with the average of 29%. The overheads are calculated for an unmodified interpreter that does not use JIT compilation. The JIT compiler is much faster on synthetic benchmarks. However the JIT shows about the same performance as the interpreter on macro benchmarks (web sites), chosen by the authors. Rajani et al [47] extend the system to support the full DOM API as well as flows via event handling. These extensions are formalized and proved sound.

Magazinius et al [38] were the first to point out in print that an inlined monitor can deal with `eval` by applying the inlining transformation to each string passed to `eval`. They prove soundness of the inlining transformation for a small imperative language with eval but lacking objects, exceptions, lambdas, dynamic access to the runtime environment, or other challenging JS features. Lack of unstructured control allows them to track PC level elegantly using lexically-scoped let-expressions rather than an explicit stack. They experiment with manual transformation of programs that use `eval`, and they implement automatic transformation for a small subset of JS. In their experiments doing the inlining manually, the inlined monitor adds an overhead of 20%–1700% depending on the browser.

Hedin and Sabelfeld [29] formalize an NSU monitor for a core subset of ECMAScript 5 and discuss IF for its various features including references, object structure, eval, and exceptions. They prove that monitoring ensures TINI and they prove the partial-correctness form of transparency: if a monitored program terminates without IF exception, then erasing security labels from its final state yields the outcome that would have been obtained by running the program without monitor. The monitor is later extended with support for browser APIs and implemented [28], though no formalization of the extension is provided. The implementation, JSFlow, is a custom JS interpreter written in JS. Like ours it can be used without modifying the browser and is comparable in language and API support. The treatment of control-flow is coarse-grained. Hedin et al [27] improve the monitor by runtime static analysis to predict potential write targets, for which the monitor upgrades labels. They prove soundness, a key point being that the monitor relies on NSU for soundness. They demonstrate, for the chosen static analysis, increases in permissiveness. The monitor is also in a position to upgrade labels at the right time, which sidesteps a complication (delayed upgrades [12]) with upgrades as code annotations.

Optimization of programs in JavaScript is tricky: even employing sophisticated program specialization techniques yields modest results. In [57] Thiemann reports that specializing JSFlow for the input, essentially yielding a compiler, gains only 1.8× speedup compared to the original.

De Groef et al [26] have built a modification of the Firefox web browser that features SME [20]. An advantage of SME is that avoids the restrictions of NSU. However, in order to avoid multiple executions of the entire software stack, the tool considers any use of web API as IO, thus treating DOM flow-insensitively. Performance overhead has been shown to be as low as 20% in IO-intensive applications (and as high as 200% on synthetic tests for a simple two-level policy), while the average memory overhead was 88%. The overhead is compared to the unmodified version of Firefox. Austin and Flanagan's SME approach has also been implemented for JS, as Firefox add-on [6]. They present a performance result, using one of the SunSpider benchmarks, that compares favorably with other SME and do not suffer as much degradation when the number of security levels increases. Soundness for these variations has been proved in [20, 6] and also in [46] which also proves transparency.

Jang et al [32] implement taint tracking for JS in web pages and use it for a large-scale empirical study of privacy violations. The monitor is implemented by rewriting and resembles ours in some respect, such as boxing and a stack of levels for indirect flow (not implicit). As the authors point out, the monitor does not track implicit flow and thus misses some information flows. They report slowdowns of 3×-8× compared to the original, depending on optimizations. Dhawan and Ganapathy [21] implement taint tracking in JS browser extensions.

Recent work has shown that taint tracking can be made sufficiently performant for use in production scenarios, offering an average of 25% overhead [39],

Complementing work on JS, Bauer et al [9] formalize and implement coarse grained taint tracking end-to-end in the Chromium browser.

Yip et al [61] implement a reference monitor called *BFlow* as a plug-in for Firefox. The monitor tracks levels of data at the grain of *protection zones* which are groups of frames where data has the same sensitivity. The data labels need to arrive from a BFlow web server. Tracking of information labels is very coarse-grained and restrictive: once a script in a frame has accessed data with a certain label, all data originating from that frame will be assigned at least that label. Consequently, no frame can handle both sensitive and public information at the same time, which prevents most useful applications and policies.

Li et al implement *Mash-IF* [34], an add-on for Firefox that uses a combination of a static data-flow analysis of a subset of JS and run-time reference monitoring of calls to DOM API to enforce information-flow policies in web mashups. This approach is less coarse-grained than BFlow, but implicit flows are not accounted for, and no soundness or security argument is made. The authors have reported that among the 10 client mashups studied, they have found no false positives and no false negatives. Policies in Mash-IF are specified by the user, utilizing a graphical interface to designate sensitivity of form elements.

Chugh et al [18] implement a hybrid monitor for JS IF tracking. To deal with dynamic code evaluation, they devise a constraint-based static analysis that applies to code with "holes" representing code that will only be determined at run time. The idea is to generate constraints on that code, to be checked at runtime before applying `eval`. The static analysis is based on a transformation that introduces taint-tracking instrumentation that is analyzed but not executed. Their staged IF verifier achieves a relatively low false positive rate of 33% in experiments on the Alexa top 100 sites, for simple but useful policies (cookies are secret, address bar is untainted, relative to the hole in which third-party code is plugged). Some important JS features are not supported (`call`, `apply`, `with`).

Sandboxing and object capability transformations are similar, though less intricate, to those of our inliner. For example, it transforms reads and writes into function calls to a run-time system as well. Performance has been reported for Caja (`http://code.google.com/p/google-caja/wiki/Performance`). In the Valija mode, which is aimed at supporting legacy ES3 code (i.e., a subset of thereof), it features slowdowns of 5× to 163×.

6. DISCUSSION

Despite decades of research on IFC, and a spate of recent work that specifically targets JS, no definitive solution has emerged. It may be that carefully engineered taint tracking will turn out to be a good compromise for general use, but there will continue to be scenarios where the assurance of strong IFC is worth its cost. In this paper we report on progress in evaluating two ideas that seem among the most promising approaches to IFC for client-side JS, namely inlining and the no-sensitive-update rule. In this paper we focus on ECMAScript 5 and parts of the Web API needed for interesting mashups and policies.

Our current prototype has too high performance overhead for many usage scenarios. However, compared with other approaches like interpreter modification [11, 33], custom meta-circular interpreters[28] or SME [26], inlined monitoring has promising paths for improving performance, notably *sparse inlining* and *extended JIT support*. Sparse inlining has been hinted at in previous work [17] and aims at reducing the amount of program instrumentation based on a static IF analysis performed ahead of inlining. The observation is that well-engineered JS programs feature substantial code fragments for which flows of information can be inferred statically. Those fragments can then remain unchanged and the rest of the program is instrumented accounting for the inferred flows. This differs from sparse labeling: in addition to allowing unlabeled values at run-time, it disables their monitoring as well. The boxing invariant still holds: the monitor core is only called from instrumented parts of code which operate on labeled/boxed values only.

In our experiments, the monitor core and API facades benefit from efficient JIT compilation, but the instrumented program itself does not. One of the likely culprits is the abundance of function calls that the program makes to the core. Many of these functions are small, pure and amenable optimization. However, it appears that the calls themselves are not optimized — especially in the recursive case. We believe it could be possible to extend the JITs with efficient support for the functional programming styles. There are precedents: `asm.js` and other fast JS subsets now have specialized compilers in many browsers. This could have a better chance of becoming a part of the mainstream engines due to the limited scope of the changes (as opposed to information-flow tracking modifications) and potential impact beyond that of our monitor. For example, it will likely benefit functional languages that have JS backends: Clojure, Haskell, Elm and others.

We build on prior works that include correctness proofs for their designs. We do not present proofs of soundness or transparency for our implementation, due to the complexity of the language and API's we are supporting. Such effort would be on the scale of a compiler verification and will be better motivated once IFC technology matures in terms of applicability and performance. However, we believe the modular structure of the inliner as well as prior work opens up interesting possibilities to simplify assurance. The rewriting rules are quite simple and the properties of the transformed program rely on those of the RTS functions as well as the isolation, confinement and mediation properties of API facades. One might formalize transparency and noninterference for the instrumented program in the mechanized framework of [13] and using the mechanized DOM formalization of [47]. To prove isolation properties of the transformed code one might leverage the existing work on verifying secure subsets of JavaScript [45]. The proof would rely on precise specifications for the RTS, which would include bisimulation-style properties for both noninterference and transparency; these could either be verified or tested.

What we really want to do is investigate NSU by experiments on sizeable existing web sites. This requires substantial additional work, to (a) implement API facades for various libraries encountered, (b) devise effective means of experimentation under realistic workloads, (c) develop precise policies for these applications, and (d) automate checking of transparency, soundness, and permissiveness.

7. ACKNOWLEDGMENTS

This work was partially supported by NSF award CNS-1228930 and by Department of Homeland Security under contract 11027-202037-DS to HRL Laboratories.

8. REFERENCES

[1] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *IEEE CSF*, 2010.

[2] A. Almeida-Matos, J. Fragoso Santos, and T. Rezk. A secure information flow monitor for a core of DOM. In *TGC*, 2014.

[3] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *IEEE CSF*, 2009.

[4] T. H. Austin and C. Flanagan. Efficient purely dynamic information flow analysis. In *ACM PLAS*, 2009.

[5] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *ACM PLAS*, 2010.

[6] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *ACM POPL*, 2012.

[7] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE Symp. Sec. & Priv.*, 2008.

[8] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. In *ESOP*, 2007.

[9] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. Run-time monitoring and formal analysis of information flows in Chromium. In *NDSS*, 2015.

[10] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Generalizing permissive-upgrade in dynamic information flow analysis. In *ACM PLAS*, 2014.

[11] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in WebKit's JavaScript bytecode. In *Prin. of Sec. and Trust (POST)*, 2014.

[12] A. Birgisson, D. Hedin, and A. Sabelfeld. Boosting the permissiveness of dynamic information-flow tracking by testing. In *ESORICS*, 2012.

[13] M. Bodin et al. A trusted mechanised JavaSript specification. In *ACM POPL*, 2014.

[14] A. Chudnov. *Inlined Information Flow Monitoring for Web Applications in JavaScript.* PhD thesis, Stevens Institute of Technology, 2015.

[15] A. Chudnov. JEST. `http://chudnov.com/jest`, 2015.

[16] A. Chudnov, G. Kuan, and D. A. Naumann. Information flow monitoring as abstract interpretation for relational logic. In *IEEE CSF*, 2014.

[17] A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *IEEE CSF*, 2010.

[18] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *PLDI*, 2009.

[19] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6), 2010.

[20] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *IEEE Symp. Sec. & Priv.*, 2010.

[21] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *ACSAC*, 2009.

[22] C. Fournet, N. Swamy, J. Chen, P.-É. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *ACM POPL*, 2013.

[23] A. Gal et al. Trace-based just-in-time type specialization for dynamic languages. In *ACM PLAS*, 2009.

[24] P. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for JavaScript. In *ACM POPL*, 2012.

[25] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *SOSP*, 2012.

[26] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *ACM CCS*, 2012.

[27] D. Hedin, L. Bello, and A. Sabelfeld. Value-sensitive hybrid information flow control for a JavaScript-like language. In *IEEE CSF*, 2015.

[28] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: tracking information flow in JavaScript and its APIs. In *ACM SAC*, 2014.

[29] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *IEEE CSF*, 2012.

[30] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP*, 1991.

[31] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. In *IEEE Symp. Sec. & Priv.*, 2013.

[32] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *ACM CCS*, 2010.

[33] S. Just, A. Cleary, B. Shirley, and C. Hammer. Information flow analysis for JavaScript. In *PLASTIC*, 2011.

[34] Z. Li, K. Zhang, and X. Wang. Mash-if: Practical information-flow control within client-side mashups. In *DSN*, 2010.

[35] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *IEEE Symp. Sec. & Priv.*, 2010.

[36] J. Magazinius, D. Hedin, and A. Sabelfeld. Architectures for inlining security monitors in web applications. In *ESSoS*, 2014.

[37] J. Magazinius, P. H. Phung, and D. Sands. Safe wrappers and sane policies for self protecting javascript. In *NordSec*, 2010.

[38] J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. In *SEC*, 2010.

[39] P. Marchenko, U. Erlingsson, and B. Karp. Keeping sensitive data in browsers safe with ScriptPolice. Univ. College London report RN/13/20, also HCSS'13.

[40] S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *IEEE CSF*, 2011.

[41] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. of Computer Security*, 14(2), 2006.

[42] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. In *ASIACCS*, 2009.

[43] K. Pingali and G. Bilardi. Optimal control dependence computation and the roman chariots problem. *ACM TOPLAS*, 1997.

[44] M. Pistoia, A. Banerjee, and D. A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *IEEE Symp. Sec. & Priv.*, 2007.

[45] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security*, 2011.

[46] W. Rafnsson and A. Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *IEEE CSF*, 2013.

[47] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer. Information flow control for event handling and the DOM in web browsers. In *IEEE CSF*, 2015.

[48] P. Ratanaworabhan, B. Livshits, and B. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *USENIX WebApps*, 2010.

[49] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of JavaScript benchmarks. In *OOPSLA*, 2011.

[50] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do - a large-scale study of the use of eval in JavaScript applications. In *ECOOP*, 2011.

[51] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *IEEE CSF*, 2010.

[52] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *ESORICS*, 2009.

[53] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. *J. Computer Security*, 17(5), 2009.

[54] J. Santos and T. Rezk. An information flow monitor-inlining compiler for securing a core of JavaScript. In *ICT Systems Security and Privacy Protection*, volume 428 of *Advances in Information and Communication Technology*. IFIP, 2014.

[55] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières. Protecting users by confining JavaScript with COWL. In *OSDI*, 2014.

[56] L. Stegeman, H. Mackenzie, et al. GHCJS: a Haskell to JavaScript compiler. `https://github.com/ghcjs/ghcjs`. Accessed August 2015.

[57] P. Thiemann. Towards specializing JavaScript programs. In *PSI*, volume 8974 of *LNCS*, 2014.

[58] J. A. Vaughan and S. Chong. Inference of expressive declassification policies. In *IEEE Symp. Sec. & Priv.*, 2011.

[59] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.

[60] D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3), 1996.

[61] A. Yip, N. Narula, M. N. Krohn, and R. Morris. Privacy-preserving browser-side scripting with BFlow. In *EuroSys*, 2009.

[62] S. A. Zdancewic. Programming languages for information security. PhD Diss., Cornell Univ., 2002.