Fission: Secure Dynamic Code-Splitting for JavaScript*

Arjun Guha[1], Jean-Baptiste Jeannin[2], Rachit Nigam[3], Rian Shambaugh[4], and Jane Tangen[5]

1 **University of Massachusetts Amherst, Amherst, MA, USA**
2 **Samsung Research America, Mountain View, CA, USA**
3 **University of Massachusetts Amherst, Amherst, MA, USA**
4 **University of Massachusetts Amherst, Amherst, MA, USA**
5 **University of Massachusetts Amherst, Amherst, MA, USA**

Abstract

Traditional web programming involves the creation of two distinct programs: a client-side front-end, a server-side back-end, and a lot of communications boilerplate. An alternative approach is to use a *tierless* programming model, where a single program describes the behavior of both the client and the server, and the runtime system takes care of communication. Unfortunately, this usually entails adopting a new language and thus abandoning well-worn libraries and web programming tools.

In this paper, we present our ongoing work on Fission, a platform that uses dynamic tier-splitting and dynamic information flow control to transparently run a single JavaScript program across the client and server. Although static tier-splitting has been studied before, our focus on dynamic approaches presents several new challenges and opportunities. For example, Fission supports characteristic JavaScript features such as `eval` and sophisticated JavaScript libraries like React. Therefore, programmers can reason about the integrity and confidentiality of information while continuing to use common libraries and programming patterns. Moreover, by unifying the client and server into a single program, Fission allows language-based tools, like type systems and IDEs, to manipulate complete web applications. To illustrate, we use TypeScript to ensure that client-server communication does not go wrong.

1998 ACM Subject Classification D.3.2. Multiparadigm Languages

Keywords and phrases JavaScript, information flow control

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.5

1 Introduction

Two decades after the introduction of JavaScript, web application security remains a challenging problem that continues to grow in significance. For example, over the past three years, the CVE database has accumulated over 2,500 cross-site scripting (XSS) vulnerabilities in popular open-source web technologies, such as Wordpress and Ruby on Rails [58, 20]. Major technology companies, such as Google, Facebook, and Microsoft regularly award bug bounties worth several thousand dollars to white-hat hackers who find vulnerabilities in their websites [25, 22, 10]. These kinds of vulnerabilities have also been making headline

* This work is supported by the U.S. National Science Foundation under grants CNS-1413985 and CCF-1408745.

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 5; pp. 5:1–5:13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

news, since hackers exploited a web-based SQL injection vulnerability to gain access to the computer systems of the U.S. Election Assistance Commission [42].

Existing Approaches. The programming languages research community has addressed the web application security problem in several different ways. For example, there is a large body of work on type systems [36, 14, 50, 35, 29, 47, 30, 5, 55, 16, 56, 57, 11] and program analyses [37, 6, 27, 15, 26, 54, 33] for JavaScript. Since many security vulnerabilities are caused by JavaScript's confounding dynamic semantics [38, 23, 46, 28, 39, 49, 45, 44, 9], a security-conscious programmer could leverage a type system or program analysis to avoid JavaScript's pitfalls. Naturally, static disciplines impose restrictions: it is difficult for tools to reason statically about dynamically-loaded code (`eval`) and other characteristic features of JavaScript. Moreover, client-side JavaScript is only half of any web application. Many security vulnerabilities occur on the server, where a variety of languages, such as PHP, Perl, Python, Ruby, and even JavaScript (using NodeJS) are in use. These other scripting languages have also been subjected to type systems and static analyses [24, 4]. However, to formally reason about the behavior and security of a web application, a tool has to consider the client-side and server-side programs together. Reasoning about multi-lingual systems is a challenging problem that is an active area of research [1, 40, 43].[1]

An alternative approach is to abandon the traditional web programming model and use a *tierless programming language*, where a single program written in a single programming language describes the behavior of the client and the server. For example, Links [18] and Ur/Web [12] provide a unified language for programming the client, server, and the database. SELinks enhances the Links model with statically-checked, label-based security policies [19]. In contrast, Swift [13] automatically partitions security-typed programs written in JIF [32] into client-side and server-side components. Tierless languages thus enable a variety of creative security solutions, but their defining characteristic is that the entire application is written in a single language, which allows programmers and developer tools to reason algebraically about their code.

JavaScript Is Hard To Let Go. Unfortunately, a key shortcoming of the aforementioned tierless languages is that they are *not JavaScript*, thus they abandon the large and vibrant JavaScript ecosystem. For example, web programmers have grown dependent on libraries like jQuery to build cross-platform web applications; modern applications use frameworks like Facebook React, which bring elements of reactive programming to the mainstream; and language extensions like JSX that allow programmers to use XML notation within JavaScript; and tools like TypeScript bring a modicum of safety to everyday JavaScript code, despite deliberately abandoning soundness. If we ask programmers to use a newly designed language, we are asking them to give up this ecosystem of libraries and language-based tools.

Our Approach. In this paper, we present Fission, a new approach to web application security that is not only compatible with the JavaScript ecosystem, but can even make existing JavaScript tools more effective. Fission allows web applications to be written as a single JavaScript program and uses dynamic information flow control and dynamic tier-splitting to automatically and securely split it into client-side and server-side code. Although

[1] Even if JavaScript is also used on the server, the client and server operate as two independent programs that communicate over HTTP. Therefore, one cannot reason about these programs using only the semantics of JavaScript.

static information flow control and tier-splitting has been explored before, we show that doing both of these *dynamically* has several advantages:

- By eschewing static approaches, Fission places no restrictions on JavaScript. Fission is fully compatible with ECMAScript 5, including characteristic JavaScript features, such as prototype inheritance and `eval`.
- We apply Fission to several large programs that use ECMAScript 6 (via Babel), JSX, and React on the client and NodeJS on the server. By fusing the client-side and server-side into a single program, Fission lets us delete a lot of brittle serialization and communication boilerplate code.
- Fission's dynamic tier-splitting allows the client and server to share code. In particular, certain higher-order function can be evaluated on either the client or the server, based on the context in which they are applied.

Beyond Security. Fission's information flow control can help programmers reason about the confidentiality and integrity of data. However, Fission's tierless design provides additional benefits. A shortcoming of the two-tier, web programming model is that it limits our ability to reason linguistically about program behavior. HTTP requests and web servers are an extra-linguistic feature and aren't part of the semantics of JavaScript. One way to address this issue is to add new language features to JavaScript [53]. However, since Fission makes web requests completely transparent, it allows us to reason about our code using just the semantics of JavaScript. Although it is rare for programmers to *reason about web programs*, there are several tools that do reason about JavaScript and these can be applied to Fission programs with minimal effort:

- We use Fission to develop server-side APIs (e.g., file I/O) for Elm [21]. Elm is a wonderful alternative to JavaScript, but without Fission, Elm programmers have to step outside the language to get any work done on the server. Furthermore, Fission could also be used with other languages that compile to JavaScript.
- We use Fission and TypeScript to write statically-typed web applications, where types ensure that client-server communication does not go wrong.
- Finally, we are able to use IDE features, such as refactoring tools, to consistently transform the client-side and server-side components of a program.

All these applications are possible because Fission faithfully implements JavaScript instead of requiring an entirely new programming language. Moreover, it is precisely because we do not design a new tierless language that Fission presents several new challenges. First, we introduce the Fission programming model in more depth.

2 A Fission Example

Figure 1 shows a two-tier program where both client and server are written in JavaScript. The purpose of this program is to display files stored on the server. When the user enters a filename and clicks "Load", the client sends a request to the server. The server either responds with the file contents or fails gracefully if the file is unreadable. Even in this trivial program, several shortcomings are apparent. First, the control-flow of the program is disjointed and bounces back and forth between the client and the server (indicated by the arrows in the figure). Second, about half the code is boilerplate needed to make requests, setup request handlers, and serialize data (highlighted in red). Finally, since the client and server are two logically distinct programs, it is difficult for programmers and tools to reason

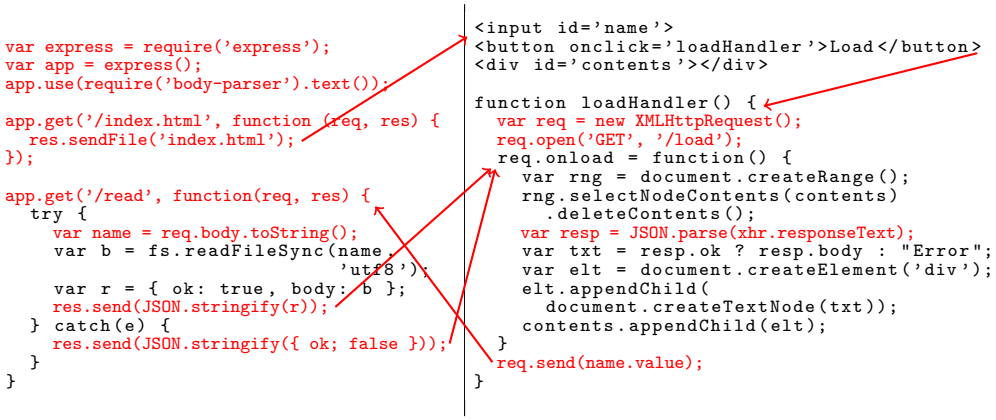

Figure 1 A canonical two-tier web application.

```
<input id='name'>
<button onclick='loadHandler'>Load</button>
<div id='contents'></div>

function loadHandler() {
  var rng = document.createRange();
  rng.selectNodeContents(contents).deleteContents();
  var txt;
  try {
    txt = declassify(fs.readFileSync(name.value));
  }
  catch(e) { txt = "Error"; }
  var elt = document.createElement('div');
  elt.appendChild(document.createTextNode(txt));
  contents.appendChild(elt);
}
```

Figure 2 A tierless version of Fig. 1.

about their behavior. JavaScript tools cannot catch the trivial bug in the figure: the client requests `/load`, but the server has a handler for `/read`.

Figure 2 refactors the program to use Fission, which addresses the problems listed above. First, the boilerplate that was highlighted in the previous figure has been eliminated, since Fission handles serialization transparently. Second, the control-flow of the program is more natural since a single function can use both client and server APIs. Finally, since we no longer have two programs that explicitly communicate over HTTP, the bug in the previous program has been eliminated. Moreover, we can now leverage JavaScript tools to manipulate the entire program without breaking client/server consistency. For example, we could use an IDE to rename the `txt` variable which is written on the server but read on the client. With Fission, JavaScript developer tools can work on the whole program and aren't limited to only the client-side code.

Attacker Model. Fission adopts the standard web attacker model [2] and assumes that an attacker can compromise the client and the network. Therefore, all values sent to the client are public to the attacker and all values received from the client are untrusted.

Information Flow Control. Fission uses dynamic information flow control, to securely partition code and data across the client and server. To a first approximation, all Fission values have two tags. A *secrecy* tag indicates whether a value is secret or public and an *integrity* tag indicates whether a value is trusted or untrusted. Therefore, Fission incorporates its attacker model as follows: Fission assumes that all client-side functions (i.e., all DOM

```
var found = 'Looking ...';
var txt = fs.readFileSync(
        '/etc/passwd', 'utf8');
if (txt.indexOf('alice') >= 0) {
  found = true;
} else {
  found = false;
}
```

Figure 3 Indirect information flow.

APIs) produce public, untrusted values and that all server-side functions (i.e., all NodeJS APIs) produce secret, trusted values.[2] Moreover, Fission will not send secret values to the client. Therefore, when a program needs to write a secret value to the client, it needs to be *declassified*. Untrusted values from the client can be *endorsed* in a similar way.

Fission asks programmers not to think about requests, responses, and serialization, but to think about the provenance of their data. This is not a new idea, but our experience with Fission shows that dynamic information flow control and dynamic tier-splitting is a powerful combination. Moreover, since Fission faithfully implements JavaScript, it supports several existing JavaScript libraries and tools with no changes required.

The Last Event Handler. Readers who enjoy functional reactive programming may be unhappy that the Fission code in Fig. 2 has one imperative event handler left. Instead of rehashing reactive programming for JavaScript, it is easy to reuse an existing JavaScript reactive programming library with Fission. Alternatively, the program could be rewritten in a language like Elm, using Fission to add support for transparent file I/O.

Overview. The rest of this paper summarizes Fission's technical approach, discusses some of the language design and implementation challenges we had to address, and some open research questions.

3 Faceted Execution

Faceted execution [7] is a form of termination-insensitive dynamic information flow control (IFC). The key idea in faceted execution is a *faceted value* (or facet for short), which is a pair of two values, where one is secret and the other is public. Which value is observed depends on the permissions of the observer. For example, when writing to the client, a facet is projected to its public component because all values on the client are visible to the attacker. Conversely, when reading from a secret file on the server, we create a facet where the private component has the file contents and the public component is a special unreadable value ($\perp$).

Let's consider a concrete example that has an indirect information flow. The program in Fig. 3 reads the user database from a server, tests if the account `alice` exists, and then sets the variable `found` to `true` or `false`. Since `readFileSync` is a NodeJS function, the variable `txt` holds a facet, where the secret component is the file contents and the public component is $\perp$, thus the client cannot directly read the file. Since the expression in the conditional uses the `txt` variable, the value of the conditional is a facet too, where the secret component is a boolean and the public component is $\perp$. Therefore, the assignment to `found` is affected by a secret value, even though no secret is directly written to `found`. Fortunately, faceted execution updates `found` to hold a facet, where the secret component is the boolean and

[2] A programmer can write more sophisticated, multi-principal policies, but these are reasonable defaults.

the public component is the original public value (the string `'Looking ...'`). Therefore, the server can observe the boolean which indicates whether the user `alice` exists, whereas the client sees the original value, thus cannot determine which branch was taken, unless the program is modified to declassify the value.

Therefore, to implement faceted execution, the control operators and primitive operations of JavaScript have to be lifted to manipulate faceted values appropriately. Facets can be nested and labeled to implement both confidentiality and integrity policies with several principals. Although Fission supports all these features, a typical Fission program only needs to reason about one principal, the server, and the programmer only needs to reason about whether values originated on the client or the server.

Cooperating Faceted Evaluators. A distinguishing characteristic of Fission is that it requires two faceted evaluators – one on the client and the other on the server – to cooperatively evaluate the program. Moreover, since the attacker model entails that the server-side evaluator cannot trust the client-side evaluator in any way, the server-side evaluator can neither send secrets to the client nor trust any information sent by the client.

Prior work on faceted execution has been based on a big-step semantics, which lends itself to a simple, direct-style interpreter. However, we had to develop a small-step faceted semantics with an explicit stack[3] because a context-switch requires an evaluator to examine its own stack, serialize it, and send it to the other evaluator. For example, if the server-side evaluator is running and the current stack frame is an application of a client-side function, at least that frame has to be serialized and transferred to the client.

It would be unsafe for the server to transfer stack frames that contain secrets to the client. So, what should happen if program applies a client-side function to a faceted argument that contain secrets? Since the Fission programming model assumes that client-side function only consume public values, a well-behaved client would simply discard the secret part of the argument and only use the public part. Instead of assuming that the client is well-behaved, the Fission server can instead project stack frames to only contain public values before transferring them to the client. This does not change the behavior of a well-behaved client, but prevents an attacker from observing secret values. Fission follows a similar approach with client-side state. The Fission programming model assumes that a web page's title, URL, cookies, etc. are always public. Therefore, when client-side state is updated to a new value, that value is projected to its public component before being transferred to the client.

Compilation and Taint Tracking. Our current implementation of Fission is an interpreter that is fast enough for interactive web pages. However, compiling Fission (or any faceted language) is challenging because each side of a facet may (or may not) follow a different branch. However, recent work [51] shows that faceted execution can be applied to taint tracking and that faceted taint tracking can be compiled in a relatively straightforward manner. Therefore, in situations where implicit flows are not a concern, a taint-tracking variant of Fission may be faster and easier to use.

4 Tier Splitting

The Fission programming model lends itself to several different implementations with a variety of tradeoffs. In particular, since the transfer of control between client and server is transparent to the program, tier splitting can be implemented in several ways.

[3] In essence, a faceted CEK machine.

Static Splitting. Although JavaScript is not statically typed, it should be possible to statically place expressions on the client or server. In brief, we could build a static control flow and data flow graph of the program, determine which expressions compute high-integrity values or consume secret values and ensure that these expressions are evaluated on the server. Swift [13] uses a richer variant of this approach, along with support for replicated data to keep the user interface responsive. We have yet to evaluate this approach in Fission, but we suspect that it may be too conservative for modern JavaScript that makes extensive use of higher-order functions. Even if programmers don't use higher-order functions themselves, they are generated by tools like Babel to implement modules. In these situations, a context-insensitive control flow graph may force too much code to needlessly run on the server. In addition, Fission supports `eval`, which gives fully static methods a lot of trouble.

Dynamic Splitting. In Fission, we tier-split the program dynamically because it produces better results for programs that use higher-order functions. The key idea is that the server can dynamically transfer control to the client (or vice versa) by transferring a prefix of the stack. However, since data sent to the client cannot contain secrets and data returned from the client cannot be trusted, the server cannot send an arbitrary portion of the stack. We use a lightweight, conservative analysis to determine which stack frames can be sent to the client without violating any of these requirements.

For example, suppose a program runs an expensive computation and displays its result on the client.

```
var x = fibonacci(50);
alert(x)
fs.writeFile('result.txt', x);
```

Since the value is not needed on the server, the expression can be evaluated entirely on the client, as long as the computation doesn't need to read secrets or update trusted values. However, if the value is also stored on the server, the computation needs to be performed on the server too.

Dynamic tier-splitting is particularly effective when a program uses higher-order functions that cannot be statically placed on either the client or the server. Consider the canonical *apply* higher-order function, which can be applied to either a function that must run on the client or a function that must run on the server.

```
function app(f, x) {
  return f(x);
}

app(fs.readFileSync, 'secret.txt');
app(window.alert, 'hello');
```

Therefore, we cannot statically determine where `app`(x) should be evaluated without considering the context where the evaluation occurs, and context-sensitive static analyses are very expensive. However, by examining the dynamic context, Fission can find a sequence of stack frames that do not read secrets, do not update trusted values, and do not execute operations like declassification and endorsement that have to be performed in a trusted context. Stack frames that meet these requirements can be evaluated on the client. These are broad requirements that allow a variety of tier-splitting mechanisms. We've developed a lightweight, conservative analysis that is effective on our benchmark programs, but it is straightforward to write contorted code that defeats the analysis and causes unnecessary context switching. A more precise analysis may be better at tier-splitting complicated code, but also have a longer running time. There is a large space of tier-splitting policies that can be explored.

5 JavaScript and Interoperability

Fission supports the ECMAScript 5.1 language standard, which is a close approximation of the JavaScript currently supported by major browsers. Unfortunately, ECMAScript is a fairly complicated language that includes getters and setters, object-oriented meta-programming features, and two language modes, in addition to well-known JavaScript pain-points, such as prototype inheritance, dynamic code-loading with `eval`, and more. Fission tackles this complexity by compiling JavaScript to a core language based on λ_{JS} [28] and S5 [45]. Fission's core language has additional features to support faceted execution as described above.

Let us now highlight Fission's implementation of `eval`, which is deeply affected by the attacker model. Note that Fission cannot implement `eval` by directly calling JavaScript's built-in `eval` function. If it did, the evaluated JavaScript code would not be able to interoperate with Fission's faceted JavaScript. Instead, Fission's implementation of `eval` builds a JavaScript AST, compiles it to a core language expression, and evaluates it using the Fission interpreter. If the call to `eval` is made in a trusted context, the JavaScript AST may even have sensitive operations like declassification and endorsement. Fission already ensures that the client is untrusted, therefore, a trusted call to `eval` may only occur on the server. In contrast, an untrusted call to `eval` may occur on either the client or the server. However, the code generated by an untrusted `eval` can neither neither access secret values nor endorse/declassify values. Notably, Fission does not need any additional mechanism to ensure that these properties hold when untrusted strings are evaluated. The same mechanisms in Fission that mediate interactions between the client and the server also ensure that untrusted strings can be safely evaluated. Although web programming best practices eschew using `eval`, it is still a commonly used construct [49], which is why put in the effort to support it.

Fission is carefully designed to support JavaScript's event-driven programming model. Therefore, a server-side event handler can transfer control to the client and vice versa. This requires full bidirectional communication, which Fission builds atop WebSockets. Moreover, if two events occur simultaneously on the client and the server, Fission serializes them to preserve JavaScript's single-threaded semantics. The current implementation of Fission suspends the client when the server is executing and vice versa, which is a reasonable default. However, there are scenarios where it is desirable to run client and server code in parallel. Other tierless languages expose server-side concurrency using libraries or special linguistic constructs [12, 18]. For the moment, we are evaluating Fission on existing NodeJS and Elm applications that do not require concurrency.

6 Applications

React-based Web applications. We have used Fission to refactor a handful of applications written for a final project in an undergraduate web programming class. All these applications use JavaScript on the client and NodeJS on the server, so refactoring involved directly calling request handlers at request sites (instead of making HTTP requests) and deleting serialization code. In addition, we had to insert declassification and endorsement operations, which is easy to do when server-side code is already factored into separate functions: we endorse all arguments and declassify the result. These applications have several hundred to two thousand lines of student-written code. However, all applications use Facebook React and several other JavaScript libraries. When they are linked together using Babel, each application has over 50,000 lines of code. Therefore, these are non-trivial examples that truly exercise the implementation. The Fission interpreter, which is written in JavaScript, takes a few seconds to load these programs, which start instantly without Fission. However, since

these are interactive programs that are not compute-heavy, the slowdown is not noticeable when they are in use.

Elm. For readers who dislike JavaScript and would rather program in a statically-typed, ML-based language, we have used Fission to implement an Elm module that adds support for reading and writing files on the server. The library requires about five lines of code for each NodeJS function exported to Elm and leverages Fission to automatically context switch between the client and server. It would be straightforward to add wrappers for more NodeJS functions to enable pure Elm applications to seamlessly run on the client and server.

TypeScript and IDEs. There exist canonical TypeScript type definitions for both the NodeJS API and the Web browser DOM API. It usually does not make sense to import both type definitions in a single program, but the TypeScript compiler does not complain if you do so. With trivial type definitions for `endorse` and `declassify`, we can write statically-typed, tierless programs using TypeScript and Fission. Moreover, we can leverage IDEs like Visual Studio which have powerful support for TypeScript programming. Without Fission, TypeScript cannot reason about the client and server code in tandem, but Fission makes it trivial for TypeScript to do so.

7 Fission for Other Languages

Although Fission is engineered to support JavaScript, it is hopefully clear to the reader that the approach is not JavaScript-specific and could be applied to other programming languages too. The most natural candidates are other dynamic languages. For example Ruby and Python are not dissimilar to JavaScript and have support programming idioms that have been challenging to statically-check [24, 4, 34]. Therefore, the Fission approach is likely to suit these languages too.

The Fission approach is useful even with certain statically typed languages. Whereas languages like Links [18], Ur/Web [12], and Swift [13] have type systems that are explicitly designed to support tier-splitting, the Fission approach can be applied to statically languages that were not designed with tier-splitting and information flow control in mind. For example, Section 6 describes we applied Fission to (the JavaScript output of) programs written in TypeScript and Elm.

8 Related Work

Fission builds on a long line of research on tierless web programming languages, some of which we've already mentioned. Fission is directly inspired by Swift [13] which uses static IFC and static analysis to partition JIF programs across the client and server. Fission's emphasis on dynamic techniques and JavaScript makes it easy for us to support reams of existing JavaScript code and presents new challenges and opportunities.

Hop.js [53] is a tierless language that is also ECMAScript-compatible and supports both NodeJS and browser APIs. Unlike Fission, Hop.js is a syntactic superset of JavaScript because it uses a special quoting syntax to explicitly demarcate the boundary between client and server code. In contrast, Fission does not change the syntax of JavaScript and thus can be used in several ways that Hop.js cannot. First, a Hop.js program cannot be consumed by ordinary JavaScript analyses and refactoring tools. Second, Hop.js cannot be used to add

server-side features to Elm (without changing the Elm compiler to generate Hop.js). Finally, Hop.js does not implement information flow control.

Fission does not syntactically distinguish client-side and server-side code, which is similar to the design of Swift and Links.[4] In contrast, in languages like Ur/Web [12], Hop [52], and Hop.js [53] it is syntactically evident when control crosses tiers. It is not clear to the authors which language design is intrinsically superior. However, since Fission does not add any new syntax to JavaScript, we get to reuse existing libraries and JavaScript developer tools.

Most web applications have three tiers: client, server, and database, and tierless languages like Links and Ur/Web unify all three tiers into a single language. Fission, with its emphasis on JavaScript, only unifies the client and the server. It is unclear if all three tiers can be unified satisfactorily for JavaScript without intentional language design (e.g., LINQ [41]).

Fission's tierless programming abstraction is built on top of an implementation of remote procedure calls [8] and distributed shared memory [3]. These abstractions require inter-machine communication, which can be optimized in several ways. For example, Remote Batch Invocation [31, 17] adds a language construct to batch several remote procedure calls together, which reduces the number of message round-trips incurred. These kinds of techniques are likely to improve Fission's performance, which currently uses very simple implementation techniques.

Fission's dynamic information flow control mechanism is based on faceted execution, which dynamically tracks implicit and explicit information flows in a fine-grained manner. However, there are alternative language-based approaches, such as Laminar [48], which is designed to make it easy to retrofit information flow control. For performance, the Laminar system requires virtual machine and operating system changes. However, Laminar's language abstractions could be adapted for JavaScript (with or without changing the VM). A possible future avenue for research may be to leverage Laminar-style "security regions" to make tier-splitting more efficient.

Acknowledgments. We thank the SNAPL'17 reviewers and our shepherd, Ranjit Jhala, for their thoughtful feedback and suggestions.

References

1 Amal Ahmed. Verified compilers for a multi-language world. In *Summit oN Advances in Programming Languages (SNAPL)*, 2015.

2 Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. Towards a formal foundation of Web security. In *IEEE Computer Security Foundations Symposium (CSF)*, 2010.

3 Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, February 1996.

4 Jong-hoon David An, Avik Chaudhuri, and Jeffrey S. Foster. Static typing for Ruby on Rails. In *IEEE International Symposium on Automated Software Engineering*, 2009.

5 Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

6 Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2014.

[4] Links supports optional placement annotations.

7 Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow control. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012.

8 Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, February 1984.

9 Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaScript specification. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014.

10 Bounty hunters: The honor roll. `https://technet.microsoft.com/en-us/security/dn469163.aspx`. Accessed Mar 24 2017.

11 Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Young-Il Choi. Type inference for static compilation of JavaScript. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2016.

12 Adam Chlipala. Ur/Web: A simple model for programming the web. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2015.

13 Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

14 Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2012.

15 Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

16 Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested refinements for dynamic languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012.

17 William R. Cook and Ben Wiedermann. Remote batch invocation for SQL databases. In *International Symposium on Database Programming Languages (DBPL)*, 2011.

18 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Formal Methods of Components and Objects*, 2006.

19 Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, label-based security enforcement for web applications. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2009.

20 CVE-2016-6316: XSS vulnerability in Action View in Ruby on Rails. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6316`. Accessed Mar 24 2017.

21 Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

22 Facebook bug bounty: $5 million paid in 5 years. `https://www.facebook.com/notes/facebook-bug-bounty/facebook-bug-bounty-5-million-paid-in-5-years/1419385021409053/`. Accessed Mar 24 2017.

23 Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2013.

24 Michael Furr, Jong-hoon David An, and Jeffrey S. Foster. Profile-guilding static typing for dynamic scripting languages. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2009.

25 Google security rewards–2015 year in review. `https://security.googleblog.com/2016/01/google-security-rewards-2015-year-in.html`. Accessed Mar 24 2017.

26 Salvatore Guarnieri and Benjamin Livshits. GateKeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, 2009.

27 Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for Ajax intrusion detection. In *World Wide Web Conference (WWW)*, 2009.

28 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010.

29 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming (ESOP)*, 2011.

30 Phillip Heidegger and Peter Thiemann. Recency types for dynamically-typed, object-based languages: Strong updates for JavaScript. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2009.

31 Ali Ibrahim, Yang Jiao an d Eli Tilevich, and William R. Cook. Remote batch invocation for compositional object services. In *European Conference on Object-Oriented Programming (ECOOP)*, 2009.

32 JIF 3.5.0: Java information flow. `https://www.cs.cornell.edu/jif`. June 2016.

33 Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A static analysis platform for JavaScript. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2014.

34 Jukka Lehtosalo. mypy. `http://mypy-lang.org`.

35 Benjamin S. Lerner, Liam Elberty, Jincheng Li, and Shriram Krishnamurthi. Combining form and function: Static types for JQuery programs. In *European Conference on Object-Oriented Programming (ECOOP)*, 2013.

36 Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. TeJaS: Retrofitting type systems for JavaScript. In *Dynamic Languages Symposium (DLS)*, 2013.

37 Magnus Madsen, Frank Tip, and Ondrej Lhoták. Static analysis of event-driven Node.js JavaScript applications. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2015.

38 Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems*, 2008.

39 Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *European Symposium on Research in Computer Security*, 2009.

40 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.

41 Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling object, relations and XML in the .NET Framework. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2006.

42 Joseph Menn. U.S. election agency breached by hackers after November vote. `http://www.reuters.com/article/us-election-hack-commission-idUSKBN1442VC`. Accessed Jan 2 2017.

43 Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. Dependent interoperability. In *Programming Languages meets Program Verification Workshop (PLPV)*, 2012.

44 Daejun Park, Andrei Stefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

45 Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, and Shriram Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In *Dynamic Languages Symposium (DLS)*, 2012.

46 Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. Adsafety: Type-based verification of javascript sandboxing. In *USENIX Security Symposium*, 2011.

47 Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Semantics and types for objects with first-class member names. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2012.

48 Donald E. Porter, Michael D. Bond, Indrajit Roy, Kathryn S. McKinley, and Emmett Witchel. Practical fine-grained information flow control using Laminar. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(1):4:1–4:51, 2014.

49 Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

50 Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015.

51 Daniel Schoepe, Musard Balliu, Frank Piessens, and Andrei Sabelfeld. Let's face it: Faceted values for taint tracking. In *European Symposium on Research in Computer Security*, 2016.

52 Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a language for programming the Web 2.0. In *Dynamic Languages Symposium (DLS)*, 2006.

53 Manuel Serrano and Vincent Prunet. A glimpse of Hopjs. In *ACM International Conference on Functional Programming (ICFP)*, 2016.

54 Ankur Taly, Úlfar Erlingsson, Mark S. Miller, John C. Mitchell, and Jasvir Nagra. Automated analysis of security-critical JavaScript APIs. In *IEEE Security and Privacy (Oakland)*, 2011.

55 Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *European Symposium on Programming (ESOP)*, 2005.

56 Peter Thiemann. A type safe DOM API. In *International Workshop on Database Programming Languages*, 2005.

57 Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Trust, but verify: Two-phase typing for dynamic languages. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015.

58 WordPress 4.6.1 security and maintenance release. `https://wordpress.org/news/2016/09/wordpress-4-6-1-security-and-maintenance-release/`. Accessed Mar 24 2017.