SANDBOXING UNTRUSTED JAVASCRIPT

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Ankur Taly
June 2013

Abstract

Many contemporary Web sites incorporate third-party content in the form of advertisements, social-networking widgets, and maps. A number of sites like Facebook and Twitter also allow users to post comments that are then served to others, or allow users to add their own applications to the site. Such third-party content often comprises of executable code, commonly written in JavaScript, that runs together with Web site's code in the user's browser. While such interweaving of codes from multiple sources often enhances the user experience, the Web site may not always trust the source of the third-party code. Moreover due to proliferation of ad-networks and content distribution networks, the true source of content may be hidden behind multiple levels of indirection.

With the rapid rise in e-commerce and social interaction on the Web, there is a vast amount of sensitive user data displayed on Web pages today — typically in the form of user-profile information, pictures, comments, credit card numbers, etc. Unless suitable restrictions are imposed, malicious third-party code executing within a Web page can easily steal or alter such sensitive information and therefore pose a significant security threat. For instance, a malicious advertisement on a page with a login form could use JavaScript to read login credentials from the form and send them to a remote server. Even worse, it could use JavaScript to define a key-logger that surreptitiously logs all user key presses and then sends this data to a (malicious) remote server.

Websites presently combat this threat by filtering and rewriting untrusted JavaScript before placing it on the page. There are a number of such JavaScript "sandboxing" tools, including Facebook FBJS, Yahoo! ADSafe, Google Caja, and Microsoft

WebSandbox. Despite their popularity, these mechanisms do not come with any rigorous specifications or guarantees. Moreover, it is even unclear what the intended security goals behind these mechanisms are.

In this dissertation, we systematically design provably-correct mechanisms for sandboxing third-party JavaScript code on Web pages. We first formally define the key security goals behind sandboxing JavaScript, using Facebook FBJS and Yahoo! ADSafe as motivating examples. We then define an operational semantics for JavaScript based on the ECMA-262 language specification, thereby establishing a mathematical basis for reasoning about JavaScript programs. To the best of our knowledge, this is the first formal semantics of the entire standards-compliant JavaScript language. Using the operational semantics, we carefully design language-based mechanisms for achieving the aforesaid security goals. We back each of these mechanisms by rigorous proofs of correctness carried out using our operational semantics. We present a comparison of our sandboxing mechanisms with Facebook FBJS and Yahoo! ADSafe, and show our mechanisms to be no more restrictive than each of them, besides having the advantage of being systematically designed and provably correct. In addition, we also uncover several previously undiscovered vulnerabilities in Facebook FBJS and Yahoo! ADSafe. These vulnerabilities have been reported to the respective vendors and the proposed fixes have since been adopted.

While language-based sandboxing mechanisms have been studied previously in the contexts of OS kernels and the Java Virtual Machine, JavaScript's lack of lexical scoping and closure-based encapsulation pose significant new challenges. We address these challenges by adapting ideas from the theory of inlined reference monitoring, capability-based security, and programming language semantics. Armed with the insights gained from designing sandboxing mechanisms, we also define a sub-language of JavaScript, called SecureECMAScript (SES), that is amenable to static analysis and wields itself well to defensive programming. We develop an operational semantics for a core subset SES-light of SES, and develop a provably-correct and fully-automated tool for reasoning about confinement properties of APIs defined in it. The language SES has been under proposal by the ECMA-262 committee (TC39) for adoption within future versions of the JavaScript standard.

Dedicated to my parents Neeta Taly and Yatindra Taly
for giving me my lifelong love for Mathematics

Acknowledgments

This dissertation would not have been possible without the excellent support and guidance that I received during the five memorable years I spent at Stanford. First and foremost, I extend my heartfelt gratitude to my advisor John C. Mitchell. John introduced me to the rich intersection of Computer Security and Formal Methods, and taught me how to view large, complicated and apparently messy systems through the lens of semantics and logic, and analyze their core structure. He provided the long-term vision and motivation for formalizing JavaScript, thereby helping me lay out the foundation of this dissertation. I have always been deeply inspired by John's astute ability in spotting research problems that are both practically relevant and theoretically interesting; it has instilled in me an appreciation for almost all areas of computer science. I hope that his taste, technique, and attitude in research continues to influence my work. Finally, I thank John for having faith in my abilities, and providing me the freedom to explore things at my own pace, while always coming to the rescue whenever I got stuck or bogged down.

I thank Sergio Maffeis for being an excellent mentor to me throughout my PhD, and collaborating with me extensively on this work. He significantly enhanced my understanding of programming language semantics and type theory, and taught me various tricks of the trade. He also provided an excellent sounding board for my research ideas, and always gave me insightful feedback (often over long late night Skype calls!).

I thank Jasvir Nagra, Úlfar Erlingsson, Mark S. Miller for being brilliant collaborators and contributing a number of ideas to my dissertation. In particular, I thank Jasvir for being such a friendly manager during my internship at Google; I still

turn to him for career advice. I thank Ulfar for helping me improve my writing and presentation skills, and providing valuable feedback on my thesis. Many thanks to Mark for enlightening me with Object Capabilities and his philosophy on computer security. I would always remember the long and stimulating conversations I had with him that greatly shaped my perspective on language-based sandboxing. Mark also helped me lay out the foundations of the SES language developed in this dissertation. Thanks to the Google PhD Fellowship program for sponsoring the last two years of my graduate studies at Stanford.

I thank Ashish Tiwari and Patrice Godefroid for being wonderful internship hosts and giving me the opportunity to work on topics outside my thesis area, thereby helping me diversify my skills. I have learned a number of techniques from them, some of which have strongly influenced the analysis methods used in this dissertation.

Many thanks to my orals committee, John Mitchell, Dan Boneh, Alex Aiken, David Mazieres and Amin Saberi for giving me feedback on my thesis defense and helping me refine my dissertation. I thank Verna Wong and Lynda Harris for shielding me from all administrative issues and making my graduate school experience completely hassle-free. I thank the Stanford security lab and my fellow batch mates: Aditya, Qiqi, Quoc and Eric for making my PhD journey so enjoyable.

Last but not the least, I thank my parents for instigating in me a love for Mathematics, for helping me obtain the best undergraduate and graduate education, and for their unconditional love and support. I thank my loving wife Preetika for always being by my side during my PhD.

Contents

List of Figures

Chapter 1

Introduction

1.1 Thesis Statement

The thesis that this dissertation supports is that it is possible to design provably-correct and practical language-based mechanisms for sandboxing untrusted third-party JavaScript on Web pages. In particular, this dissertation presents sandboxing mechanisms for mediating access to critical hosting page resources (such as the DOM and other browser services), and for isolating one third-party JavaScript component from another. The design and proof of correctness for the mechanisms is based on a formal operational semantics for JavaScript, also defined in this dissertation. We justify the practical usefulness of our mechanisms by comparing them with widely used real-world sandboxing mechanisms such as Facebook FBJS [82] and Yahoo! ADsafe [15]. In addition to spotting multiple security-vulnerabilities in both FBJS and ADsafe, our analysis shows that our mechanisms are strictly more permissive.

1.2 What is JavaScript?

JavaScript is an HTML scripting language that is widely used for client-side programming within Web pages. It was designed by Brendan Eich in 1995 at Netscape for the purpose of adding dynamic capabilities to Web pages. Prior to the advent of JavaScript, Web pages essentially consisted of static text and images structured using

HTML. JavaScript code allows Web pages to dynamically alter content, resize windows, verify information within forms, react to user events, etc. Over the last decade, JavaScript has gained tremendous popularity with almost all pages embedding it in some form. It is therefore dubbed as the "assembly language of the Web".

JavaScript is an object-based language, with support for prototype-based inheritance, first-class and higher-order functions, and a built-in eval function that allows dynamic generation of code. It is designed for both programmers and non-programmers, with a very relaxed coding discipline. For instance, JavaScript is tolerant to minor errors such as missing semi-colons, missing variable-declarations, and various arity and type mismatch errors. JavaScript also provides other flexibilities to the programmer, such as the ability to add, delete and modify properties of an object, dynamically generate object property names from strings and dynamically modify all built-in objects. While from a code analysis/review standpoint these flexibilities would be viewed as *bugs*, most JavaScript programmers view them as *features* as they greatly lower the learning curve for novice programmers and allow for rapid prototyping.

JavaScript code is embedded in a Web page by directly placing the source code within a ⟨script⟩... ⟨/script⟩ tag or inside certain special attributes of HTML elements, as in ⟨form onclick="alert('Hello!')"⟩, which alerts a "Hello" message whenever the form is submitted. JavaScript from third-party sites can be loaded by using the tag: ⟨script src=*third-party-URL*⟩. For a particular frame, all third-party JavaScript codes loaded as part of this mechanism share the same execution environment, thus allowing them to compose freely. This mechanism is the key enabling feature for third-party JavaScript advertisements that are pervasive on almost all Web pages today.

JavaScript code manipulates features of a Web page by accessing its Document Object Model (DOM) of the page which is essentially a tree representation of the various elements of the page. The nodes of the tree are called DOM objects, examples of which include the the window object which is always the root of the DOM tree, the document object which represents the main HTML document, the location object which represents the URL of the page, the cookie object which represents the cookie associated with the page, etc. DOM objects can be programmatically accessed using

JavaScript and provide various methods for traversing and modifying the DOM tree. For instance, document.forms provides access to the various ⟨form⟩ elements present on a page, document.getElementById(...) allows selecting DOM objects by their identifier, document.location.replace(...) allows replacing the current URL with a new one (thereby navigating the page), and so on. We refer the reader to [86] for an introduction to the DOM.

Standardization. Due to the widespread success of JavaScript, in November 1996 it was handed over to ECMA International [1] for standardization. The standards body released the first edition of the language-specification, called ECMA-262, in June 1997. The ECMA-262 specification did not include a model for the DOM and was designed to cover the core JavaScript language implemented in all browsers that were current back then. The DOM was standardized separately by the World Wide Web Consortium (W3C) [86]. Since June 1997, multiple editions of the ECMA-262 specification have been released, with the most recent being the 5^{th} edition [33] which was released in Dec 2009. When the work for this dissertation began in January 2008, the specification was in its 3^{rd} edition [32]. Strangely the 4^{th} edition of the specification was never released due to disagreements within the ECMA-262 committee (TC39). We call the language captured by 3^{rd} edition of the specification as ECMAScript3 (ES3), and that captured by the 5^{th} edition as ECMAScript5. The 5^{th} edtion also defines a "strict mode" which we call ECMAScript5-strict (ES5-strict). In this dissertation we study the languages ES3 and ES5-strict.

1.3 The JavaScript Sandboxing Problem

While JavaScript was originally designed for adding small scripting capabilities to Web pages, its uses have become dramatically more sophisticated during the past 15 years. JavaScript content on Web pages today provides rich functionalities such as — autofilling of form fields, reacting to user events (like key presses and mouse clicks), asynchronously interacting with the server, providing social-networking support, etc., which together lead to an enhanced user experience. Furthermore, a large amount of JavaScript content these days comes from third-party sources such as advertisement

networks, map services (such as Google Maps) and social networking websites (such as Facebook). A classic example of this is the website Yelp [2] which combines static information about restaurants, hotels and shopping malls with content from Google Maps to provide detailed search results. In general, such applications obtained by composing content from different sources are known as *mashups*. The individual contents being composed are called *components* of the mashup and content providers are called *principals*.

While a Google Maps component on a page is considered trusted and assumed to only access the portion of the page's DOM that is assigned to it, such an assumption cannot be made about other third-party JavaScript components embedded on the page (e.g., advertisements that get loaded from unknown servers after multiple points of indirection). Since the DOM provides access to cookies, form fields, the location bar, and other sensitive elements, untrusted JavaScript components pose a significant security threat to the hosting page. For instance, a malicious advertisement on a page could use JavaScript to read the site's cookies from the DOM and compromise the user's session. Moreover, since all third-party JavaScript components included directly using the ⟨script src=*third-party-URL*⟩ tag run in the same execution environment, they can potentially write into each other's variables and as a result compromise each other's execution. For instance, a malicious advertisement can modify the contents of a benign advertisement such that all clicks on the benign advertisement get directed to it. Thus untrusted JavaScript components not only pose a security threat to the hosting page but also to other JavaScript components running on the page. With the tremendous growth of e-commerce on the Web, protecting Web pages from such threats has become an increasingly important concern for both application developers and end users.

In summary, a fundamental problem for all Web pages embedding untrusted third-party JavaScript code is to develop a mechanism for *sandboxing* the third-party code in order to: (i) protect critical resources belonging to the hosting page, and (ii) protect one third-party component from another. This is called the *JavaScript Sandboxing* problem. We note that the term "sandboxing" was originally coined by Wahbe in his seminal work on software fault isolation [87], in order to describe a technique for

isolating faults within untrusted OS Kernel extensions. However over the years the term has been used for abstractly describing techniques for restricting the behavior of untrusted programs. The objective of this dissertation is to develop techniques for solving the JavaScript sandboxing problem. In solving the problem we intend to respect one important constraint which we discuss next.

Constraint: No browser modification. The JavaScript Sandboxing problem can be solved by implementing a reference monitor within the browser that enforces a security policy on third-party code. Since the browser has complete visibility into the execution of JavaScript, this approach can be used to enforce fine-grained access-control and information-flow control policies on third-party JavaScript code. In the past few years there have been a number of proposals [36, 63, 54] based on this approach. However in this work, we impose the constraint that the proposed sandboxing mechanism should not require browser modification. The motivation behind this constraint is to enable our mechanism to have an impact on existing Web applications in the short term. In order to respect backwards compatibility, browser vendors are severely constrained in the modifications that they can make to existing browsers. Moreover, since there are multiple browsers today with significant market shares, a mechanism involving browser modification would have to be adopted by all the major browser vendors. While mechanisms involving browser modifications could become viable in the long term, perhaps by including the mechanism within browser standards, the short-term viability still remains restricted. In this dissertation we therefore restrict ourselves to techniques that work on all existing browsers and can be implemented easily by hosting pages.

1.4 Two Broad Approaches

In this section we discuss two broad approaches for solving the JavaScript sandboxing problem that do not require browser modification.

1.4.1 IFrame-Based Sandboxing

The first class of techniques [44, 39, 7, 34] rely on using *IFrames* for sandboxing third-party JavaScript code. IFrame, which stands for "Isolated Frame", is a browser mechanism for providing an isolated execution environment within a page. Content can be loaded within an IFrame using the HTML tag ⟨iframe src=*third-party-URL*⟩. IFramed JavaScript code runs in its own global scope and has its own DOM sub-tree. Furthermore, all browsers support the *same-origin policy* which ensures that code running in IFrames from different origins cannot access each other's DOM trees.

Given the above features, one approach to solving the sandboxing problem is to place all third-party JavaScript components in different IFrames. By definition IFrames would provide variable separation and the same-origin policy would provide DOM isolation. Unfortunately, this approach turns out to be too restrictive for many contexts as it completely prevents any client-side communication between third-party code and the hosting page. The hosting page, for instance, may wish to expose special APIs to third-party code that provide mediated access to certain hosting page DOM objects. Recognizing this limitation, since June 2008, all major browsers started providing an additional API called postMessage that allows communication of string-based messages between IFrames. Various researchers and Web application developers have since then built abstractions on top of postMessage for facilitating richer communication (see [7, 44]). The main upsides of the IFrames+postMessage technique are that it is easy to deploy and it does not require inspecting or modifying third-party code.

In spite of the advantages, IFrames+postMessage is still not the preferred approach in many settings. IFrames limit content to a specific portion of the page (confined within a boundary), and therefore cannot be used to embed higher-revenue ads that intend to float over the entire hosting page. Besides, the hosting page has no control over code running inside an IFrame and therefore cannot enforce a presentation specification on it. This lack of control also has security implications — malicious IFramed code could potentially exploit browser vulnerabilities (say in the JavaScript parser) and compromise critical components of the browser.

The pros and cons described above are analogous to those of process-based isolation in the context of operating systems. IFrames correspond to processes and the postMessage mechanism corresponds to the inter-process communication channel. The alternative approach for isolating untrusted modules in operating systems is *language-based isolation* which we discuss next. Just as OS inter-process isolation is useful in some situations while others require language-based isolation between lightweight threads in the same address space, we expect both IFrames and language-based isolation to be useful in future Web applications. Furthermore, as browsers evolve, IFrames may be able to enforce fine-grained restrictions on code and thus overcome their present limitations.

1.4.2 Language-Based Sandboxing

An alternative approach to IFrame-based sandboxing, is to pre-process third-party code, applying filters and source-to-source rewriting, and then directly place it in an appropriately initialized execution environment within the hosting page. This approach is called *Language-based sandboxing*. Prominent examples of sandboxing mechanisms based on this approach include Facebook FBJS, Yahoo! ADsafe, Google Caja, and Microsoft Web Sandbox.

Facebook [81] is a leading social networking site that allows third-party JavaScript applications to be embedded within user profile pages and interact with trusted Facebook libraries. In order to protect users from malicious applications, Facebook requires all applications to be written in a subset of JavaScript called FBJS. Facebook then suitably rewrites all FBJS applications to insert certain run-time checks and make them run in what is intended to be a "separate namespace". Yahoo! ADsafe [15] is a mechanism for sandboxing third-party advertisements on an advertisement network. ADsafe statically analyzes third-party code and only allows it to interact with a special ADsafe provided library. Google Caja is a framework for sandboxing third-party applications on platforms such as OpenSocial [21], iGoogle [24], and the Yahoo!'s Application Platform [93]. Caja uses a compilation process that takes JavaScript code and produces code in Cajita, a well behaved capability-safe subset of

JavaScript. Compiled Cajita applications from different principals can be safely composed by allocating them capabilities that only allow controlled interaction between the applications. Microsoft Web Sandbox is a framework designed for restricting third-party scripts within Web pages and also emails. It involves rewriting third-party code in order to inline a reference monitor that enforces a policy on all object property and method accesses. Other language-based mechanisms for sandboxing untrusted JavaScript are described in [26, 70, 67, 52]

While there is a long history of language-based sandboxing techniques in the contexts of operating system kernels [87, 62, 65, 73] and the Java Virtual Machine [23, 89, 88, 19, 18, 17], the main challenge in adapting those techniques to JavaScript lies in dealing with the various non-standard features of JavaScript like the lack of lexical scoping, lack of closure-based encapsulation (in most implementations), and the presence of dynamic code generation. Surprisingly, even simple properties of code such as the set of accessible global variables are impossible to compute statically in JavaScript. The main goal of this work is to combat these challenges and systematically develop language-based mechanisms for sandboxing third-party JavaScript code.

1.5 Our Approach

In this dissertation, we define a formal operational semantics for standardized JavaScript and use it to systematically design language-based sandboxing mechanisms. The semantics is subsequently also used to prove correctness of the mechanisms designed. The architecture of our mechanism and the restrictions enforced by it are inspired from that of Facebook FBJS and Yahoo! ADsafe.

When we started this work in January 2008, our original goal was to formally prove isolation properties of FBJS and ADsafe. However in trying to do so, we discovered problems in both systems. The version of ADsafe that was current back then did not properly account for definitions that might occur on a hosting page. For instance the hosting page could provide aliases to eval and other dangerous constructs, which were not shielded away from third-party JavaScript code. While analyzing

FBJS we found that some of the runtime checks enforced by the rewriting could be circumvented, thereby allowing applications to obtain direct references to critical DOM objects. We also subsequently discovered that the Facebook variable-renaming algorithm was not semantics-preserving due to corner cases involving object-property access via variables. Based on the subtlety of these errors, and others that may exist in similar systems, we decided to systematically design our own mechanism for enforcing the same security goals as intended by FBJS and ADsafe.

We analyzed the implementations of FBJS and ADsafe, and found that both of them have a common security goal and enforcement architecture. The security goal is to ensure that all access to DOM objects is mediated by a trusted reference monitor, and the execution of one third-party component does not influence the execution of another. The common enforcement architecture, which we call API+Language-Based Sandboxing (API+LBS), consists of two key steps. In the first step the hosting page code implements an API that provides mediated access to critical hosting page resources. In the second step, the hosting page server fetches the third-party code, *filters* and *rewrites* it such that it can *only* access the API and cannot influence the execution of other third-party JavaScript component. This rewritten code is then directly embedded on the hosting page. The foundations of the API+LBS architecture lie in the *object-capability* theory of securing systems (see [58, 42]). The methods of the API are *capabilities* supplied to third-party code and the sandboxing mechanism is the *loader* that loads third-party code with the given set of capabilities [58].

It is easy to see that the security of a sandboxing mechanism based on the API+LBS architecture relies on the correct functioning of the API and the language-based sandbox. Typically, API implementations are closely tied to the access policy enforced on critical hosting page resources, with a fresh policy requiring a fresh implementation. The language-based sandbox on the other hand needs to be constructed only once for the language in which third-party code is written, regardless of the API implementation. In this work, we therefore consider the problems of *designing* a secure language-based sandbox, and *verifying* that a given API implementation confines access to critical hosting page resources. We call these the *Sandbox Design* and *API Confinement* problems respectively.

Informally, the sandbox design problem requires designing a language-based mechanism that ensures the following properties: (1) (*Hosting page Isolation*) Third-party components obtain access to critical hosting page resources *only* via the API, and (2) (*Inter-component Isolation*) No third-party component can influence the execution of another third-party component . The API confinement problem requires verifying that no sandboxed third-party code can use the API to obtain a direct reference to a critical hosting page resource. In other words, no interleaving of the API method calls ever leads to a reference to a critical resource. If API methods are viewed as capabilities, then the API Confinement problem is also known as the *Overt Confinement Problem for Capabilities* [41].

1.6 Thesis Contributions

The main contribution of this dissertation is in solving the Sandbox Design and API Confinement problems for standardized JavaScript. In particular, we present solutions to the Sandbox Design problem for ECMAScript3 (ES3) and the API Confinement problem for a sub-language of ECMAScript5-strict (ES5-strict). We were unable to solve the API Confinement problem for ES3 due to its lack of lexical scoping, lack of closure-based encapsulation and the presence of dynamic code generation facilities that together make static analysis very challenging. To overcome these ES3 limitations, we identified a sub-language SES of ES5-strict hat is both amenable to static analysis and also provides support for defensive programming. The language SES has been under proposal[1] t by the ECMA-262 committee (TC39) for adoption within future versions of the JavaScript standard. We formalized a core subset SES-light of SES and solved the API Confinement problem for SES-light. In what follows, we describe the key contributions made in each chapter of this dissertation.

Chapter 2 provides an overview of the languages ES3 and ES5-strict. Chapter 3 presents a survey of Facebook FBJS and Yahoo! ADsafe. It then provides a detailed

[1]SES was originally conceived by Mark S. Miller who also led the proposal in the ECMA committee (TC 39). The first formal characterization of SES was developed by Ankur Taly and Mark S. Miller.

description of the API+LBS architecture, the Sandbox Design problem and the API Confinement problem.

Chapter 4 defines a small-step operational semantics for the complete ES3 language. Unlike previous formalizations that focused on small core subsets of JavaScript, this semantics models the entire ES3 language as described in the standard. It is therefore suitable for reasoning about JavaScript code "in the wild," which is crucial for security analysis.

Chapters 5 and 6 present language-based mechanisms for solving the Sandbox Design problem for ES3. Chapter 5 considers the special case of sandboxing a single third-party component, and Chapter 6 considers the general case of sandboxing mashups consisting of multiple sequentially composed third-party components. All of these mechanisms are backed by a proof of correctness carried out using the operational semantics of ES3. Both chapters also present a comparison of the mechanisms designed to FBJS and ADsafe, along with a number of previously undiscovered security vulnerabilities found in FBJS and ADsafe during the course of this work.

Chapter 7 presents the sub-language SES of ES5-strict and its key advantages over ES3. It then presents a small-step operational semantics for a core subset of SES called SES-light. Chapter 8 presents a Datalog-based static analysis technique for solving the API Confinement problem for SES-light. The soundness of the technique is formally established using the operational semantics of SES-light developed in Chapter 7. The chapter also describes an implementation of the technique in the form of an automated tool `ENCAP`, and discusses a previously undiscovered vulnerability in the Yahoo! ADsafe DOM API found using `ENCAP`. Finally, it describes a repaired version of the API for which `ENCAP` successfully proves DOM confinement.

One of the challenges in writing this dissertation was that all languages and systems analyzed during the course of this research have been a moving target. While all the major browsers release a new version roughly every six months, the mechanisms Facebook FBJS and Yahoo! ADsafe are updated nearly continuously. As a result we have organized the main technical chapters in a chronological order. Chapters 4, 5, 6 are based on ECMAScript3 and describe research carried out from January 2008 to May 2010. Chapters 7, 8 are based on ES5-strict and describe research carried out

from June 2010 to Dec 2011. The versions of FBJS analyzed in Chapters 5 and 6 are ones that were current in March 2009 and November 2009 respectively. The versions of ADsafe analyzed in Chapters 6 and 8 are ones that were current in November 2009 and November 2010 respectively.

Chapter 2

An Overview of Standardized JavaScript

JavaScript was originally developed in 1995 at Netscape as an HTML scripting language. Due to its widespread popularity it was soon implemented by all the leading browser vendors. In November 1996, JavaScript was handed over to ECMA International for standardization. The ECMA committee released the first language specification, called ECMA-262, in June 1997, and named the standardized language "ECMAScript". Since then multiple versions of ECMAScript have been released by the ECMA committee. In this dissertation, we study the languages: (i) ECMAScript3, which is ECMAScript based on 3^{rd} edition [32] of the ECMA-262 specification (released in December 1999), and (ii) ECMAScript5-strict, which is the "strict mode" of ECMAScript based on 5^{th} edition [33] of the ECMA-262 standard (released in December 2009). In the rest of this chapter, we provide a brief overview of some of the core features of the languages ECMAScript3 and ECMAScript5-strict.

2.1 The Language ECMAScript3

The main primitives of ECMAScript3 (ES3) are first-class and potentially higher-order functions, dictionary like objects which can be constructed using object expressions (without the need of class declarations), prototype-based inheritance, dynamic

typing, implicit type conversions, and a built-in eval operation. In what follows we describe each of these briefly using illustrative examples. We use the prompt ES3> to denote a hypothetical strictly ECMA-262, 3^{rd} edition compliant interpreter.

Objects and functions. Objects in ES3 are essentially records mapping strings to values. They can be created by calling a constructor function in the context of a new operator, as in var o = new Object(), or by using a literal expression $\{pn_1{:}e_1,\ldots,pn_n{:}e_n\}$ as shown by the following example

```
ES3> var o = {bar: 10, foo: 5}
```

In the above code, variable o stores a first-class reference to the object created. All object references in ES3 are unforgeable. Fields of an object are known as *properties,* which can be accessed using the "dot" notation, as in x.bar, or using the indexing operator [ ] , as in x["bar"]. The latter case allows for dynamic generation of property names. Object properties can be added, updated or deleted dynamically as illustrated by the following example:

```
ES3> o.baz = 1; o["bar"] = 10; delete o.foo ;
% adds property "baz" initialized to 1, sets property "bar" to 10 and deletes property"foo"
```

Such dynamic additions and deletions of property names makes static typing of objects very challenging for ES3. ES3 supports anonymous and higher-order functions, which are represented as first-class objects with updatable properties. For example, the statement

```
ES3> var foo = function() {return function() {return 0;};}
```

creates an anonymous function object and stores a reference to it in the variable foo. Besides anonymous creation, functions can also be created using the function declaration syntax, as in

```
ES3> function foo() {return 42;}
```

which declares a variable foo in the current scope that holds a reference to the corresponding function object. Function object references can be assigned to object properties, effectively making them methods of the object. For example:

```
ES3> o.baz = function() {return 42;};
ES3> o.baz(); % returns 42
```

Prototype-based inheritance. Every object in ES3 has a prototype object from which it inherits properties. While looking up a property in an object, if the property is absent then it is looked up in the prototype-object and then the prototype of the prototype-object and so on. This chain of prototypes is commonly called the "prototype chain" and it always terminates with the built-in object Object.prototype. The built-in object Object.prototype provides access to pre-defined functions such as "toString", "hasOwnProperty", "isPrototypeOf", etc. All function objects have another built-in object Function.prototype in the last-but-one position of the prototype chain. Surprisingly, the properties of such built-in objects are also redefinable.

The prototype of an object gets fixed at the time of construction to the "prototype" property of the constructor. For example:

```
ES3> var foo = function() {this.foo = 1;};
ES3> var op = {bar: 10};
ES3> foo.prototype = op;
ES3> var o = new foo(); % sets the prototype of o to op.
ES3> o.bar; % result: 10.
```

For objects created via the literal expression, the prototype is Object.prototype. In ES3 it is not possible to directly get hold of the prototype of an object. However as we will see in later in this section, most browsers implementations of the ES3 specification allow direct access to the prototype object via a special "__proto__" property.

Scopes and identifier lookup. Scopes in ES3 are defined lexically, however stack frames are represented using first-class objects instead of conventional variable environments. The properties of these objects correspond to the variables declared in the particular scope. Scope objects created during a function call are called "Activation objects" and have a special internal status. All execution begins within the global scope, which is represented by a built-in object called the global object. All global variables are therefore properties of the global object. Identifier resolution is carried out by traversing down the stack of scope objects and looking for a property with

the same name as the identifier. This is not as straightforward as it seems as property lookup also involves traversing the prototype chain. Due to this unconventional design, scopes cause a lot of confusion amongst programmers. For instance, in the following example

```
Object.prototype.x = 42;
ES3> var x = 24;
ES3> var foo = function() {return x;};
ES3> foo(); % result: 42.
```

during the call to foo, the identifier x resolves to the property "x" of the prototype (Object.prototype) of the current scope object.

To complicate scoping further, ES3 supports a construct called with which allows for placing arbitrary objects on top of the current scope stack, as illustrated by the following example.

```
ES3> var o = {foo:1};
ES3> with(o) {foo = 3} % sets property "foo" of o to 3.
```

Finally, although ES3 has block statements such as if(...){ ... }, for(...){...}, these statements do not create fresh scopes. Scopes are only created by function blocks. For example:

```
ES3> function foo() {
       if(true) {
         var x = 24;
         function bar() {var x = 42; return x;}
       }
       return x;
     };
ES3> foo(); % result: 24.
```

The call foo() returns 24 since the identifier x in the return statement binds to the var x declaration inside the if block which has the same scope as the body of foo.

Keyword this. In most object-oriented languages, the keyword this is used by

object methods to refer to their containing object. However in ES3, object methods essentially contain references to function objects and multiple objects could have methods holding references to the same function objects. As a result the semantics for this are highly non-standard. Furthermore, they have also changed significantly between the ES3 and ES5-strict specifications. Below we discuss the semantics as supported by the ES3 specification.

The this argument gets passed implicitly during all method and function calls. For a method call o.foo(...), the this argument is the object o. For a function call foo(...), the this argument is the scope object in which the identifier foo gets resolved, except when the scope object is also an activation object in which case the this argument is the global object. For example, the this argument in the nested call to bar in the code

```
ES3> var foo = function() {
                 var bar = function() {return this;};
                 return bar();}
ES3> foo(); % result: global object.
```

is the global object as the identifier bar resolves to an activation object (corresponding to call to foo). The idiosyncratic semantics of this is often exploited by malicious code in order to obtain a reference to the global object and thereby maliciously alter security-critical global variables.

Dynamic code generation. ES3 provides certain built-in functions which allow dynamic conversion of strings to code. For example:

```
ES3> eval{"o.foo␣=" + "1"};
```

updates the property foo of object o to 1. Here eval is a built-in function object reachable from the property "eval" of the global object. Another such built-in function that allows dynamic code generation is the Function constructor. The existence of eval and Function in ES3 makes static analysis very challenging as some of the code that executes may not even exist statically.

Types. Values in ES3 are associated with one of the following runtime type tags: "object", "function", "boolean", "string", "number", "undefined". The type tag of a value can be obtained using the typeof operator. For example:

```
ES3> typeof 1; % result: "number".
ES3> typeof function() {return 42;}; % result: "function".
```

Type tags are used to determine whether an operation is allowed on a value. For example, a function call foo() involves checking that the type tag associated with foo is "function". If the check fails then the operation throws a TypeError. In certain other cases a type mismatch leads to an implicit type conversion. For example in the expression o[p], if p is not a string, then it is implicitly converted to a string. The conversion semantics vary depending on the type tag associated with p. As an example, if p is a number then it is converted to a string by surrounding it with apices " ". If p is an object then it is converted to a string by invoking the "toString" method of the object. We leave the discussion on the semantics of such conversions to Chapter 4.

Reflection. Reflective capabilities pervade ES3. Most of these capabilities are obtained via methods inherited from built-in prototype objects. For instance, the "hasOwnProperty" method inherited from Object.prototype allows code to determine if an object has a particular "own" property, that is, a property present directly in the object structure.

```
ES3> var o = {a: 42};
ES3> o.hasOwnProperty("a"); % result: true.
```

The method "call" inherited by function objects from Function.prototype allows invoking a function with an arbitrary this argument

```
ES3> var foo = function() {return this};
ES3> var o = foo.call({a:42}); % passes the first argument as the this argument to foo.
ES3> o.a % result: 42.
```

ES3 also has a special looping construct for(p in o){...} which allows looping over all properties of an object. For example the following code creates a concatenation of the names of all "own" properties of the object o.

```
ES3> var s = "";
ES3> var o = {a:1, b:2, c:3};
```

```
ES3> for (p in o) s = s + p; % stores "abc" in s.
```

There are many more such reflective capabilities supported by ES3, all of which are covered in our formal operational semantics (described in Chapter 4). As we will see in the next section, most ES3-based browsers provide even more reflective capabilities, such as the ability to get a reference to the prototype of object and the ability for a called function to get a reference to its caller.

2.1.1 Implementation Extensions

All browser implementations of ES3 support a number of other features beyond the ones specified in the standard. Furthermore, browser implementations also deviate from the standard in the implementation of many existing constructs. While we leave the discussion of these deviations to Chapter 4, in what follows we discuss three additional constructs supported by most browser implementations of ES3: *setters and getters*, the "__proto__" property, and the "caller" property. We use the prompt JS> for browser implementations of ES3.

Setters and getters. A *getter* for an object-property is essentially a function that gets invoked when the property is read, and a *setter* is a function that gets invoked when the property is written. Setters and getters allow a programmer to override the normal property access and assignment functions. Most browser implementations of ES3, use the syntax {get p(){... }, set p(){... },... } for creating getters and setters. The following example illustrates the semantics:

```
JS> var o = { cnt:0,
              val:0,
              get p() {this.cnt = this.cnt + 1; return val;},
              set p(x) {this.val = x;}};
JS> o.p; % result:0.
JS> o.p = 2;
JS> o.p; % result:2.
JS> o.cnt; % result: 1.
```

Property "__proto__". According to the ES3 specification, it is not possible for user-level code to obtain a direct reference to the prototype of an object. However most browser implementations of ES3 define a property named "__proto__" in all objects that allows for getting and setting the prototype of the object. Thus one can write the following code.

```
JS> var o = {a: 24};
JS> var p = {b: 42};
JS> o.b; % result: undefined.
JS> o.__proto__ = p;
JS> o.b; % result: 42 (obtained from the prototype p).
```

Property "caller". Perhaps the most unusual extension to ES3 supported by browsers is the "caller" property. All function objects have a property named "caller" which during function invocation stores a reference to the immediate *caller* function according to the runtime call graph, or null if called at the top-level. The following example illustrates this behavior.

```
JS> var foo = function() {return foo.caller;};
JS> var bar = function(x) {return x();};
JS> bar(foo); % result: reference to bar since it is the caller for foo.
```

2.2 The Language ECMAScript5-strict

In December 2009, the ECMA committee released the 5^{th} edition [33] of the ECMA-262 standard which includes a "strict mode" that is a syntactically and semantically restricted subset of the full language. Shifting from normal to strict mode is done by mentioning the "use strict" directive at the beginning of a function body, as in function(){"use strict" ; ...}. In this dissertation, we analyze the strict mode subset as a separate programming language ECMAScript5-strict (ES5-strict) and assume that all code runs under a global "use strict" directive. ES5-strict is essentially a subset of ES3, with the addition of *setters and getters* and some new built-in functions. Setters and getters were modeled based on the then-current browser implementations

Restriction	**Property enforced**
No delete on variable names	Lexical Scoping
No prototypes for scope objects	Lexical scoping
No with	Lexical scoping
No this coercion	No ambient access to global object
Safe built-in functions	No ambient access to global object
No "callee", "caller" properties on arguments objects	Closure-based encapsulation
No "caller", "arguments" on function objects	Closure-based encapsulation
No arguments and formal parameters aliasing	Closure-based encapsulation

Figure 2.1: ES5-strict restrictions over ES3

that already supported them. Amongst the new built-in functions added, the most interesting is Object.freeze which take an object as an argument and *freezes* it — make all its properties immutable and prevents any further property additions and deletions.

In the remainder of this section, we discuss the key syntactic and semantics restrictions imposed by the ES5-strict on top of ES3. Figure 2.1 summarizes the restrictions along with the language properties that holds as a result. These properties serve as the main motivation behind imposing the restrictions. Below, we discuss each of the properties and the corresponding restrictions in detail, and argue why the properties fail for ES3 and hold for ES5-strict.

Lexical scoping. The presence of prototype chains on scope objects (or activation records) and the ability to place first-class objects on the scope stack, makes a lexical scope analysis of variable names unsound. In fact, it is impossible to statically determine the binding declarations of variables. This makes ordinary renaming of bound variables (α-renaming) unsound and significantly reduces the feasibility of static analysis. Consider the following code as example.

```
ES3> Object.prototype[<e>] = 24;
ES3> var x = 42;
```

```
ES3> var f = function foo() {return x;};
ES3> f();
```

It is impossible to decide statically if the identifier x on the third line binds to the declaration on the second line. This is because if the evaluation of expression e returns "x" then the the identifier x does *not* bind to the declaration on the second line, and otherwise it does. Similar corner cases arise when code can potentially delete a variable name or can use the with construct to artificially place user objects on the scope stack. Recognizing these issues, ES5-strict forbids the use of the with construct, and deletion of variable names. Furthermore, the semantics of ES5-strict models scope objects (or stack frames) using the traditional environment record data strcuture and therefore without any prototype inheritance.

Safe closure-based encapsulation. As discussed in the previous section, ES3 implementations in most browsers support the "caller" property, that provides callee code with a mechanism to access its caller function. This breaks closure-based encapsulation, as illustrated by the following example. Below, a trusted function takes an untrusted function as argument and checks possession of a secret before performing certain operations.

```
ES3> function trusted(untrusted, secret) {
       if (untrusted() === secret) {
        % process secretObj
       }
     }
```

Under standard programming intuition, this code should not leak secret to untrusted code. However the following definition of untrusted enables it to steal secret.

```
ES3> function untrusted() {return arguments.caller.arguments[1];}
```

ES5-strict eliminates such leaks and make closure-based encapsulation safe by explicitly forbidding implementations from supporting the properties "caller", "arguments" on function objects.

No Ambient Access to Global Object. ES3 provides multiple (and surprising)

ways for code to obtain a reference to the global scope object, which is the root of the entire DOM tree and hence security-critical in most browser implementations. For instance, the following program can be used to obtain a reference to the global object.

```
ES3> var o = {foo: function () {return this;}}
ES3> g = o.foo;
ES3> g(); % result: global object.
```

This is because the this value of a method when called as a function gets coerced to the global object. Furthermore, methods "sort", "concat", "reverse" of the built-in Array.prototype object and method "valueOf" of the built-in Object.prototype object also return a reference to the global object when invoked with certain ill-formed arguments. ES5-strict prevents all these leaks and only allows access to the global object by using the keyword this in global scope, or by using any host-provided aliases such as the global variable window.

Chapter 3

Key Problems

In this chapter, we survey two prominent sandboxing mechanisms: Facebook FBJS and Yahoo! ADsafe. We identify a common enforcement architecture used by both of them, called API+Language-based-sandboxing (API+LBS), and then define two key problems: *Sandbox Design* and *API Confinement*, that need to be solved while designing sandboxing mechanisms based on this architecture. The rest of this dissertation focusses on solving these problems.

3.1 Existing Sandboxing Mechanisms

We describe some of the core features of Facebook FBJS and Yahoo! ADsafe. We particularly focus on security-relevant features and ignore implementation details which often vary across different versions.

3.1.1 Facebook FBJS

Facebook [81] is a web-based social networking application. Registered and authenticated users store private and public information in their Facebook profiles (stored on the Facebook servers), which may include personal data, list of friends (other Facebook users), photos, and other information. Users can share information by sending messages, directly writing on a public portion of a user profile, or interacting with

Facebook applications.

Facebook applications can be written by any user and are deployed in two ways: as external web pages displayed within a nested frame in the user profile, or as integrated components of a user profile. Integrated applications are very popular, as they provide a richer user experience and affect the way a user profile is displayed.

Since Facebook applications are in general untrusted, arbitrary JavaScript code included as part of an integrated application could pose a significant security risk to the user. In particular such JavaScript code could access critical portions of the page's DOM, steal cookies and navigate the page to malicious sites. As a result integrated applications are sandboxed on the Facebook server before including them on a user's profile. The design of the sandboxing mechanism is intended to allow application developers as much flexibility as possible, while protecting user privacy and site integrity.

Sandboxing mechanism. Facebook requires all JavaScript content present within integrated applications to be written within FBJS, which is a fragment of ES3 designed to restrict applications from accessing arbitrary parts of the DOM of the containing Facebook page. The source application code is checked to make sure it contains valid FBJS, and some rewriting is applied to limit the application's behavior before it is rendered in the user's browser.

While FBJS has the same syntax as JavaScript, a preprocessor consistently adds an application-specific prefix to all top-level identifiers in the code, with the intention of isolating the application's namespace from the namespace of other parts of the Facebook page. For example, the expression `document.domain` is rewritten to `a12345_document.domain`, where `"a12345_"` is an application-specific prefix. Since this renaming prevents application code from directly accessing most of the host and native JavaScript objects (e.g., the `document` object), Facebook provides libraries that are accessible within the application's namespace. For example, a special library object is stored in the variable `a12345_document`, that mediates interaction between the application code and the true `document` object.

Additional steps are taken to restrict the use of the special identifier `this` in FBJS code. This is because the expression `this`, executed in the global scope, evaluates

to the window object, which is the global scope itself. An application could simply use an expression such as this.document to break the namespace isolation and access the document object. Since renaming this would drastically change the meaning of JavaScript code, occurrences of this are replaced with the expression ref(this), which calls the function ref to check what object this refers to and accordingly returns null if it refers to window, and behaves as the identity function otherwise (see Chapter 5 for further discussion of ref and the revised version $FBJS.ref that is presently used).

Other indirect ways of getting hold of the window object involve accessing certain standard or browser-specific predefined object properties such as "__parent__" and "constructor". Therefore, FBJS blacklists such properties and rewrites any explicit access to them to an access to the useless property "__unknown__". For property accesses of the form o[e], where the property name is dynamically generated by evaluating expression e, FBJS rewrites that access to a12345_o[idx(e)] where the function idx enforces a blacklist on the result of e (see Chapter 5 for further discussion of ref and the revised version $FBJS.idx that is presently used). Finally, FBJS code is barred from using with and is run in an environment where methods such as "valueOf" of the Object.prototype object, which may be used to access (indirectly) the window object, are redefined to something harmless.

3.1.2 Yahoo! ADSafe

Many web pages display advertisements, which are typically produced by untrusted third parties (online advertising agencies) unknown to the publisher of the hosting page. Even an advertisement as simple as an image banner is often loaded dynamically from a remote source by running a piece of JavaScript provided by the advertiser or some (perhaps untrusted) intermediary. Hence, it is important to isolate web pages from advertising content, which may potentially consist of a malicious script.

The ADsafe mechanism proposed by Yahoo! is designed to allow advertising code to be placed directly on the host page, limiting interaction by a combination of static analysis and syntactic restrictions. As explained in the documentation [15], "*ADsafe defines a subset of JavaScript that is powerful enough to allow guest code to perform*

valuable interactions, while at the same time preventing malicious or accidental damage or intrusion. The ADsafe subset can be verified mechanically by tools like JSLint so that no human inspection is necessary to review guest code for safety.". The high–level goal of ADsafe is to "*block a script from accessing any global variables or from directly accessing the DOM or any of its elements*".

Sandboxing Mechanism. Concretely, the ADsafe mechanism consists of two components: (1) An ADsafe library that provides restricted access to the DOM and other page services, and (2) A static filter JSLint that discards untrusted JavaScript code if it makes use of certain language constructs like the expression this, statement with, expression o[*e*], identifier or properties names beginning with "_", etc. The goal of the filter is to ensure that JavaScript code that passes through it only accesses security-critical objects by invoking methods on the ADsafe library. The ADsafe library provides various methods that allow safe access to DOM objects. It is designed with the goal that all DOM objects stay confined within the library and third-party code never obtains a direct reference to any of them.

According to the design of ADsafe, all third-party advertisement code must be written using ADsafe specific programming idioms, otherwise they would get discarded by JSLint. For example, the JavaScript code

```
var location = document.location;
```

that accesses the DOM, should be written by the user as

```
var location = ADSAFE.get(document,"location");
```

where ADSAFE.get is a library method that only allows dynamic lookup of non-blacklisted properties of objects.

3.2 API + Language-Based Sandboxing Architecture

In the previous section we described the design of Facebook FBJS and Yahoo! ADsafe. Interestingly, we find that while the implementations of the mechanisms are different,

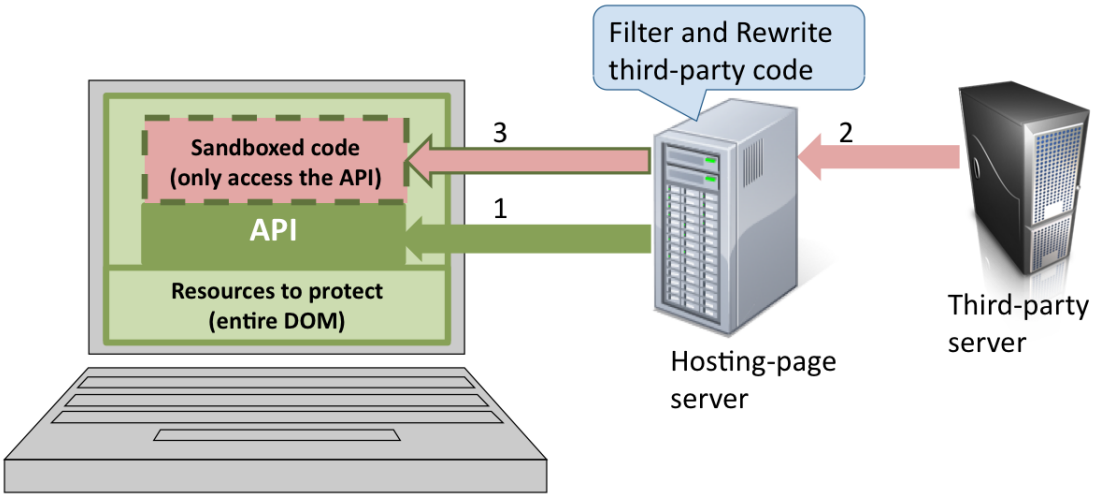

Figure 3.1: API+Language-based-sandboxing architecture

both of them share a common abstract security goal and enforcement architecture. In what follows we describe this security goal and architecture.

Security Goal. FBJS and ADsafe have two broad security goals:

- (**Isolation**) Execution of one third-party component must not influence the execution of another third-party component.
- (**Mediated Access**) All access to the DOM and other page services must be mediated by a trusted reference monitor

The above goals together ensure that sandboxed third-party components embedded on hosting pages cannot maliciously access DOM objects or impact other third-party components. Perhaps unsurprisingly the above security goals are consistent with the *Memory protection* and *Secure services* goals of traditional language-based sandboxing mechanisms developed in the contexts of operating system kernels and the Java virtual machine [88, 20, 73].

Architecture. The API+Language-Based-Sandboxing (API+LBS) architecture is described in Figure 3.1. The numbers on the arrow indicating the order in which code is loaded. We assume that the hosting page has a set of critical resources that it wants to protect, presumably the DOM objects.

In the first step the hosting page implements an API that provides mediated access to critical resources. In the case of FBJS and ADsafe, the APIs are the respective DOM libraries included on the page. In the second step, third-party code is fetched by the hosting page server, and then *filtered* and *rewritten*. The filtering and rewriting together is called *Language-based sandboxing*. In principle, filtering may range from simple syntactic checks to full-fledged static analysis, with obvious tradeoffs between efficiency and precision. For instance, ADsafe carries out a full-fledged static analysis using JSLint while FBJS only checks for certain syntactic patterns. In some cases, the sandboxing step may also involve prepending the third-party code with an initialization code that installs *wrappers* around critical resources present in the execution environment. As an example, FBJS installs a wrapper around the "valueOf" method of Object.prototype so that it never returns a reference to the global object. Finally, if there are multiple third-party components, then they are all fetched, sandboxed, and then embedded on the hosting page in sequence.

The goal of the API is to implement a reference monitor that provides mediated access to critical hosting page resources. The goal of the language-based sandboxing is ensure the following properties: (1) (*Hosting page Isolation*) Third-party components obtain access to critical hosting page resources *only* via the API. (2) (*Inter-Component Isolation*) No third-party component can influence the execution of another third-party component. It is easy to see that inter-component isolation helps in achieving the overall Isolation goal. Hosting page isolation on the other hand is necessary for achieving the overall Mediated Access goal. This is because the API is ineffective in enforcing mediated access if third-party code is able to bypass it and directly obtain a reference to a critical resource. Hosting page isolation is responsible for guaranteeing the absence of such "bypass" mechanisms.

The foundations of the API+LBS architecture lie in the *object-capability* theory of securing systems (see [58, 42]). The methods of the API are *capabilities* supplied to third-party code and the sandboxing mechanism is the *loader* that loads third-party code with the given set of capabilities [58]. We now illustrate by an example, some of the subtleties involved in designing sandboxing mechanisms based on the API+LBS architecture.

Example. Consider a hosting page with a log data structure that holds references to critical objects. The page wants to embed third-party code with the restriction that it can *only* write to the log. A mechanism for enforcing this restriction can be designed using the API+LBS architecture. The hosting page first creates an API object that only has method "push" which allows data to be pushed on to the log. This object is then provided to third-party code by placing it in a global variable api.

```
var priv = criticalLogArray;
var api = {push: function(x){priv.push(x);}};
```

Third-party code is filtered and rewritten such that it can *only* access the global variable api. A necessary requirement for establishing correctness of this mechanism is to verify that the API object itself does not leak a direct reference to criticalLogArray, as that would allow reading data from the log. While the example above may suggest that this requirement can be easily verified, the addition of the following "store" method to the API may suggest otherwise:

```
api.store = function(i,x){priv[i] = x;};
```

Although a cursory reading shows that neither API methods "push" or "store" return a reference to the criticalLogArray, the API fails to confine the reference. Third-party code may gain direct access to the reference by calling methods of the API and mutating external state, as in the following code:

```
var result;
api.store('push',function() {result = this[0]});
api.push(); % result holds reference to criticalLogArray.
```

The exploit makes unanticipated use of the "store" method by supplying "push" as the first argument instead of a numeral. A possible fix would be to modify the "store" method so that it coerces its index argument to a number.

```
api.store = function(i,x){priv[i+0] = x;}; % i+0 implicitly converts i to a number.
```

This example demonstrates that the correctness of an API+LBS based mechanism relies not only on restricting third-party code to the API but also verifying that the API confines access to critical hosting page resources.

3.3 Security Problems

We now state two key problems: *Sandbox Design* and *API Confinement* that need to be solved while designing sandboxing mechanisms based on the API+LBS architecture. The objective of this dissertation is to solve both these problems for standardized JavaScript.

3.3.1 Sandbox Design Problem

As mentioned in Section 3.2, the goal of the language-based sandbox is to restricts all third-party components to the API (Hosting page Isolation), and also enforces isolation *between* various components (Inter-component Isolation). We call the conjunction of these two isolation goals as *Mashup Isolation*. For ease of analysis, we first consider the problem of only enforcing hosting page isolation and then move to problem of enforcing mashup isolation.

Hosting page isolation. The goal here is to ensure that third-party code obtains access to critical hosting page resources only via the API. In general, the set of critical resources consists of certain browser-defined host objects and/or certain DOM objects, such as window, document, etc. In the initial JavaScript execution environment, all DOM and other browser-defined host objects are stored within certain pre-defined global variables, called *host global variables*. Without loss of generality, we assume that the API is implemented as a set of objects (methods of the objects being the API functions) stored in certain global variables, called *white-listed global variables*. By design, the set of white-listed global variables is disjoint from the set of host global variables. Therefore enforcing hosting page isolation boils down to solving the following problem

> **Hosting Page Isolation Problem.** Given a white-list $\mathcal{G}$ of global variables, design a filtering and rewriting mechanism for third-party code that ensures that it can only access global variables named in the white-list $\mathcal{G}$

In Chapters 5 and 8 we solve this problem for ES3 and a sub-language of ES5-strict respectively.

Mashup Isolation. Isolating mashups requires enforcing inter-component isolation in addition to hosting page isolation. In this work we only consider a simple form of mashups obtained by sequentially composing multiple third-party components. Such mashups are called *basic mashups*. While hosting page isolation can be enforced by restricting all third-party components to *only* access white-listed global variables, inter-component isolation can be enforced by ensuring that one component does not write to a portion of memory that another component reads from. This essentially means is no communication channel through memory between any two mashup components. The problem of isolating mashups can be informally stated as follows.

> **Mashup Isolation Problem.** Given a white-list $\mathcal{G}$ of global variables, design a filtering and rewriting mechanism for a basic mashup that ensures the following: (1) (*Hosting page Isolation*) each mashup component only accesses global variables named in the white-list $\mathcal{G}$, and (2) (*Inter-component Isolation*) execution of one mashup component does not involve writing to a memory location that another component reads from.

In Chapter 6 we solve this problem for a semantically-restricted form of ES3.

3.3.2 API Confinement Problem

The goal of the API in the API+LBS architecture is to implement a reference monitor that mediates *all* access to critical hosting page resources. A necessary condition for achieving the complete mediation goal is that API must never leak a direct reference to a critical resource. In other words, no interleaving of the API method calls must ever lead to a reference to a critical resource. We call this condition *API Confinement*. We verify confinement for an API by considering all *usages* of the API that are allowed to sandboxed third-party code. Here by "sandboxed code" we refer to code that can only access the API, in other words, code that respects hosting page isolation. Thus the problem of verifying API confinement can be informally stated as follows.

> **API Confinement Problem.** Give a set of security critical resources L_{sec} and an API implementation, verify that no sandboxed third-party code can use the API to obtain a direct reference to a resource in L_{sec}

If API methods are viewed as capabilities, then the *API Confinement problem* is also known as the *Overt Confinement Problem for Capabilities* [41]. In Chapter 8, we define a provably-sound and automated technique for solving the API Confinement problem for a sub-language of ES5-strict.

Chapter 4

An Operational Semantics for ECMAScript3

In this chapter[1], we present an overview of our operational semantics of ES3 [45], based on the 3rd edition of the ECMA-262 standard [32]. Our semantics has two main parts: one-step evaluation relations for the three main syntactic categories: expressions, statements and programs of the language, and definitions for all the built-in objects that are meant to be provided by an implementation. In the process of developing the semantics, we examined a number of perplexing (to us) ES3 program situations and experimented with a number of ES3 implementations. To ensure accuracy of our semantics, we structured many clauses after the ECMA-262 specification [32]. As a validation of the semantics we prove progress and preservation theorem for well-formed ES3 programs.

Need for a formal semantics. The central goal of this dissertation is to design and analyze language-based sandboxing mechanisms for restricting untrusted third-party JavaScript, such as those that have arisen recently in connection with online advertising and social networking [15, 11, 3, 82]. While analyzing the security properties of these systems, it is important to consider attacks that could be created using arbitrary JavaScript, as opposed to some subset used to develop the trusted application. It is

[1]This chapter is based on joint work with Sergio Maffeis.

therefore important to have a formal model that can be used to reason about JavaScript code *in the wild*. Although there have been scientific studies of limited subsets of the language [5, 83, 94], there appears to be no previous formal investigation of the entire ES3 language, on the scale defined by the informal ECMA specification [32]. In this work we therefore develop a small-step operational semantics for JavaScript that covers the language addressed in the 3rd edition of the ECMA-262 standard [32]. As we will see later in this chapter, ES3 is full of unconventional features such as the ability to make first-class objects as stack frames (using with), hoisting of variable and function declarations before function invocation, implicit type conversions, and extensive reflective capabilities. As a result manual reasoning of even small snippets of ES3 code is extremely challenging. A formal semantics provides a systematic way of understanding and reasoning about ES3 programs.

Organization. The rest of this chapter is organized as follows: Section 4.1 discusses the scope of the semantics and some of the challenges involved in defining it. Section 4.2 describes the actual semantics and Section 4.3 discusses deviations between the semantics and various browser implementations. Section 4.4 presents a formal framework for reasoning about ES3 programs and states the progress and preservation theorems for well-formed ES3 programs. Finally, Section 4.5 presents related work.

4.1 Challenges and Scope

As described in Chapter 2, ES3 supports functional programming with anonymous functions, which are widely used to handle browser events such as mouse clicks. ES3 also has objects that may be constructed as the result of function calls, without classes. The *properties* of an object, which may represent methods or fields, can be inherited from a prototype, redefined or even removed after the object has been created. This makes it conceptually possible to represent activation records by objects, with properties corresponding to assignable variables of the current scope. Static typing for full ES3 is also challenging, because it is possible to change the value of a property arbitrarily, or remove it from the object. ES3 also has eval that allows user-provided strings to be parsed and evaluated as code, thereby hindering static typing

further. For these and other reasons, formalizing ES3 and proving the correctness of security mechanisms designed for ES3 poses substantial challenges. Below we discuss a few unconventional features from ES3 that are challenging to model in a formal semantics.

One example feature of JavaScript that is different from other languages is the way that declarations are processed in an initial pass before bytecode for a function or other construct is executed. Some details of this phenomenon are illustrated by the following code:

```
ES3> function f() {
       if (true) {
         function g() {return 1;}
       } else {
         function g() {return 2;}
       }
       function g() {return 3;}
       return g();
       function g() {return 4;}
     }
```

This code defines a function f whose behavior is given by one of the declarations of g inside the body of f. However, different implementations disagree on which declaration determines the behavior of f. Specifically, a call f() should return 4 according to the ECMA specification. However Spidermonkey 1.7.0 (hence Firefox 2.0) returns 4, Rhino 1.7R11.7 returns 1, and JScript 7.0 (hence Internet Explorer 7.0) returns 2. Intuitively, the function body is parsed to find and process all declarations before it is executed, so that reachability of second declarations is ignored. Given that, it is plausible that most implementations would pick either the first declaration or the last. However, this code is likely to be unintuitive to most programmers.

Those with some curiosity may also enjoy the following example on the difference between a declaration function f(...){ ... } and the expression var f = function (...){ ... } which uses another form of binding to associate the same name with an apparently

equivalent function.

```
ES3> function f(x) {
       if ( x == 0)
         return 1;
       return f(x−1);
     }
ES3> var h = f;
ES3> h(3); % result: 1.
ES3> function f(x) {
       if ( x == 0)
         return 3;
       return x∗f(x−1);
     }
ES3> h(3); % result: 6.
```

Unsurprisingly, the call to h(3) after the second line evaluates to 1. However, the call to h(3) after the third line produces 6. In effect, the call to h(3) first executes the first body of f, apparently because that's the declaration of f that was current at the place where h was declared. However, the recursive call to f in the body of line one invokes the declaration on the third line!

A number of other features of ES3 provide additional challenges for development of a formal semantics for the language. We list some of them below:

- *Redefinition.* Values of built-ins undefined, NaN and Infinity, and especially Object, Function and so on can be redefined. Therefore the semantics cannot depend on fixed meanings for these predefined parts of the language.

- *Implicit mutable state.* Some JavaScript objects, such as Array.prototype are implicitly reachable even without naming any variables in the global scope. The mutability of these objects allows apparently unrelated code to interact.

- *Property Enumeration.* JavaScript's for in loop enumerates the properties of an object, whether inherited or not. The ECMA specification [32] does not define

the order of enumeration of properties in a for in loop, leading to divergent implementations.

- *this confusion.* JavaScript's rules for binding this depend on whether a function is invoked as a constructor, as a method, or as a normal function. If a function written to be called in one way is instead called in another way, its this property might be bound to an unexpected object or even to the global environment.

Beyond this dissertation. Our framework for studying the formal properties of ES3 closely follows the specification document and models all the features of the language that we have considered necessary to represent faithfully its semantics. The semantics can be modularly extended to *user-defined getters and setters*, which are part of JavaScript 1.5 but not of the ECMA-262 standard. We believe it is similarly possible to extend the semantics to interface with *DOM objects*, which are part of an independent specification (a formal subset is presented in [22]), and are available only when JavaScript runs in a Web-browser. However, we leave development of these extensions to future work.

For simplicity, we do not model some features which are laborious but do not add new insight to the semantics, such as the switch and for construct (we do model the for in construct), parsing (which is used at run time for example by the eval command), the built-in Date and Math objects, minor type conversions like ToUInt32, etc., and the details of standard procedures such as converting a string into the numerical value that it actually represents. For the same reason, we also do not model *regular expression matching*, which is used in string operations.

4.2 Operational Semantics for ES3

Our small-step operational semantics for ES3 covers all constructs described in the 3rd-edition of the ECMA-262 standard, and closely follows the structure of the standard. Because of the complexity of JavaScript and the number of language variations, our semantics is approximately 70 pages of rules and definitions, formatted in ASCII.

The rules are expressed in a conventional meta-notation, that is not directly executable in any specific automated framework, but is designed to be humanly readable, and a suitable basis for rigorous but un-automated proofs.

In principle, for languages whose semantics are well understood, it may be possible to give a direct operational semantics for a core language subset, and then define the semantics of additional language constructs by expressing them in the core language. Instead of assuming that we know how to correctly define some parts of JavaScript from others, we decided to follow the ECMA-262 specification as closely as possible, defining the semantics of each construct directly as given in the specification. While giving us the greatest likelihood that the semantics is correct, this approach also prevented us from factoring the language into independent sub-languages.

Given the volume of the entire semantics, we only describe the main semantic functions and some representative axioms and rules here; the full semantics is currently available online [45].

4.2.1 Syntax

Figures 4.1, 4.2 and 4.3 present the entire syntax of top-level (user-level) ES3 programs. Following the ECMA-262 specification, we divide the syntax into values, expressions, statements and programs. In the grammar, we abbreviate $t_1, \ldots, t_n$ with $\tilde{t}$ and $t_1 \ldots t_n$ with t^* (t^+ in the nonempty case). Furthermore, $[t]$ means that t is optional, $t \mid s$ means either t or s, and in case of ambiguity we escape with apices, as in "[" t "]". While defining the grammar, we also follow systematic conventions about the syntactic categories of metavariables, to give as much information as possible about the intended type of each operation. For conciseness, we use short sequences of letters to denote metavariables of a specific type. For example, m ranges over strings, pv over primitive values, etc.

The syntax for variables and values is shown in Figure 4.1. Variables are strings as usual with the exception of certain reserved words such as this, delete, etc. Values are either pure values or references. A pure value (va) is either a primitive value (pv), a heap location (l) or the special value @null. Primitive values are standard with three

special values: @NaN, which ironically is a special number called "Not a Number", @Infinity and @undefined. The value @NaN is the result obtained when certain arithmetic operations are applied to numeric and non-numeric values. For instance, "a" $*$ 42 $=$ @NaN. Similarly @undefined is a value that is returned on certain failing operations, such as reading a non-existent property of an object. Heaps locations are prefixed by the symbol #. They include certain constant heap locations #global, #obj, ... etc which correspond to built-in objects. Fresh heap locations range over #1, ... References are a special kind of internal values which are pairs of nullable locations (ln) and strings (m). They are generated as a result of evaluating expressions. If the location part of a reference is @null then the reference is called a *null reference*

We denote thrown exception values by enclosing them within $\langle\rangle$, as in $\langle va \rangle$. As a convention, we append w to a syntactic category to denote that the corresponding term may belong to that category or be an exception. For example, lw denotes an address or an exception.

The syntax for expressions, statements and programs is shown in figures 4.2 and 4.3 respectively. In the grammar PO, UN, and BIN range respectively over primitive, unary and binary operators. Our partitioning of the syntax into expressions, statements and programs is based entirely on the ES3 specification. Finally, in order to keep the semantic rules concise, we assume that source programs are legal ES3 programs, and that each expression is disambiguated (e.g. $5 + (3 * 4)$).

Since the semantics is small step, it also introduces new expressions and statements in the program state for book-keeping. Such terms are called *internal terms* and are prepended with the symbol "@" in order to distinguish them from the user-level syntax of ES3. Most of the internal expressions and statements correspond to the internal functions defined in the ES3 specification (chapter 8, [32]). The full semantics, defined in [45], gives the entire grammar for internal expressions and statements as well. In the next section, we will elaborate on a few of these internal terms as we describe the semantic rules. Henceforth we will use *terms* to denote the union of all user-level and internal expressions, statements and programs.

x ::=	foo | bar | ...	identifiers (excluding reserved words)
v ::=	$va \mid r$	values
va ::=	$pv \mid ln$	pure values
pv ::=	$m \mid n \mid b$ | @undefined	primitive values
m ::=	"foo" | "bar" | ...	strings
n ::=	$-n$ | @NaN | @Infinity | 0 | 1 | ...	numbers
b ::=	true | false	booleans
l ::=	#global | #obj | #objproto | ...	locations
	#1 | ...	
r ::=	ln*m	reference
ln ::=	l | @null	nullable addresses
w ::=	$\langle va \rangle$	exception

Figure 4.1: Syntax for ES3 variables and values

e ::=	this	this object
	x	identifier
	pv	primitive value
	"[" $[\tilde{e}]$ "]"	array literal
	$\{[p\tilde{n}{:}e]\}$	object literal
	(e)	parenthesis expression
	$e.x$	property accessor
	e_1 "[" e_2 "]"	member selector
	new $e_1[([\tilde{e_2}])]$	constructor invocation
	$e_1([\tilde{e_2}])$	function invocation
	function $[x]$ $([\tilde{x}])[P]$	[named] function expression
	e PO	postfix operator
	UN e	unary operators
	e_1 BIN e_2	binary operators
	$(e_1$? $e_2{:}e_3)$	conditional expression
	$(e_1{,}e_2)$	sequential expression
pn ::=	$m \mid n \mid x$	property names

Figure 4.2: Syntax for ES3 expressions

$s ::=$	$\{s*\}$	block
	var $[(x[\tilde{=}e])]$	assignment
	e	expression
	if (e) s [else s]	conditional
	while (e) s	while
	do s while (e)	do-while
	for $(e$ in $e)$ s	for-in
	for (var $x[=e]$ in $e)$ s	for-var-in
	continue $[x]$	continue
	break $[x]$	break
	return $[e]$	return
	with(e) s	with
	x:s	label
	throw e;	throw
	try$\{s\}$ [catch(x)$\{s_1\}$] [finally$\{s_2\}$]	try
	s_1;s_2	sequence
	;	skip
$P ::=$	$fd \mid s \mid s\ P \mid fd\ P$	programs
$fd ::=$	function x $([\tilde{x}])\{[P]\}$	function declarations

Figure 4.3: Syntax for ES3 statements

$L ::=$	L:$l \mid emp$	stack
$cl ::=$	fv, L	closures
$fv ::=$	function$([\tilde{x}])\{P\}$	function values
$ov ::=$	$va\{[\tilde{a}]\} \mid cl$	object values
$a ::=$	ReadOnly | DontEnum | DontDelete	attributes
$p ::=$	m | @prototype | @class | @call | @construct | @closure | @value | @this	properties
$o ::=$	$\{p$:$\tilde{ov}\}$	objects
$H, K ::=$	$\tilde{l:o}$	heaps
$t ::=$	$e \mid s \mid P$	terms

Figure 4.4: Heaps and Stacks in ES3

4.2.2 Building Blocks

Heaps and Stacks. The complete definitions of Heaps and Stacks are present in Figure 4.4. Heaps (H) map locations to objects, which are records from objects properties (p) to object values (ov). Properties can be strings or certain internal property names which are prepended by the symbol "@". Object values are either pure values, with certain optional attributes (a), or function closures. Attributes indicate certain restrictions on property access. A closure is a pair of a function value (fv) and an execution stack which we describe next.

As discussed in Chapter 2, the ES3 specification models variable environments for program scopes using objects. Furthermore, certain constructs (such as with) allow for placing arbitrary objects on top of the current execution stack. Therefore in our effort to be closely conformant to the specification, we model execution stacks as a list of objects whose properties represent bindings of local variables in the scope. Formally a stack is a list L of locations. The empty stack is denoted by emp. The ES3 specification uses the term "scope-chains" for "stacks", and in our description we will use both these terms interchangeably. We use $Heaps^{ES3}$ and $Stacks^{ES3}$ as the universe of all possible $ES3$ heaps and stacks respectively.

Types. ES3 values are dynamically typed. The internal types are:

$$T ::= \text{Undefined} \mid \text{Null} \mid \text{Boolean} \mid \text{String} \mid \text{Number} \mid \text{Object} \mid \text{Reference}$$

Types are used to determine conditions under which certain semantic rules can be evaluated. Given a value v, we assume a function $Type(v)$ that returns the internal type of v.

Helper Functions. The semantics makes use of a standard set of helper functions to manipulate heaps. $alloc(H, o) = H_1, l$ allocates object o at a fresh location l and returns the location along with the resulting heap H_1. $H(l) = o$ retrieves object o from location l in heap H. $o.p = va\{[\tilde{a}]\}$ gets the value of property p of object o, along with the possibly empty list of attributes. $H(l.p \leftarrow ov) = H_1$ sets the property p of object at location l to the object value ov, and returns the resulting the heap H_1. $del(H, l, p) = H_1$ deletes property p from the object at location l in heap H. $p \in o$

co ::=	(ct, vae, xe)	completion type
ct ::=	Normal | Break | Continue | Return | Throw	ct-type
vae ::=	@empty | va	ct-value
xe ::=	@empty | x	ct-identifier

Figure 4.5: References and Completion types in ES3

holds if object o has a property p.

Reduction Rules. Our small-step operational semantics for ES3 consists of a universe of program states Σ and a set of state reduction rules $\mathcal{R}^{ES3}$. The semantics is formally denoted by $(\Sigma, \mathcal{R}^{ES3})$. Program states are triples H, L, t consisting of a heap H, stack L and term t. The general form of a state reduction rule is

$$\frac{\langle premise \rangle}{H_1, L_1, t_1 \rightarrow H_2, L_2, t_2}$$

The rule specifies that state H_1, L_1, t_1 can be reduced to state H_2, L_2, t_2 if $\langle premise \rangle$ holds. In our semantics, we have three small-step reduction relations $\xrightarrow{e}$, $\xrightarrow{s}$, $\xrightarrow{P}$, depending on whether the term part of a state is an expression, statement or program respectively.

The evaluation of an expression returns a value v or an exception w. The evaluation of a statement or program terminates with a special internal value called completion value (co), as defined in Figure 4.5. A completion value is triple consisting of a type, value and identifier. The type specifies the kind of termination, the value specifies the value obtained on evaluation (or it is @empty) and the identifier specifies the program point where execution must proceed to next (or it is @empty). The value of a completion is relevant when the completion type is Return (denoting the value to be returned), Throw (denoting the exception thrown), or Normal (propagating the value to be returned during the execution of a function body). The identifier of a completion is relevant when the completion type is either Break or Continue, denoting the program point where the execution flow should be diverted to. If the type of a completion is Normal then the completion is called *normal* else it is called *abrupt*.

The state reduction relations $\xrightarrow{e}, \xrightarrow{s}, \xrightarrow{P}$ are recursive, and mutually dependent. The semantics of programs depends on the semantics of statements which in turn

depends on the semantics of expressions which in turn, for example by evaluating a function, depends circularly on the semantics of programs. These dependencies are made explicit by contextual rules, that specify how a transition derived for a term can be used to derive a transition for a bigger term including the former as a subterm. For instance, the evaluation of the statement return e requires evaluation of the expression e. This dependency is formalized using the following rule:

$$\frac{H, L, e \xrightarrow{e} H_1, L_1, e_1}{H, L, sCe[e] \xrightarrow{s} H_1, L_1, sCe[e_1]}$$

Here sCe is a statement context for evaluating expressions. Examples of such contexts include return _, with(_)s, and so on (see [45] for a complete list). Therefore if $H, L, e \xrightarrow{e} H_1, L_1, e_1$ then $H, L,$ return $e \xrightarrow{s} H_1, L_1,$ return e_1. Furthermore, if the inner sub-expression evaluates to an exception then that exception must be propagated to the top level. This is formalized by the following contextual rule:

$$H, L, sCe[w] \xrightarrow{s} H, L, w$$

Transition axioms (rules that do not have transitions in the premises) specify the individual transitions for basic terms (the redexes). For instance, for return statements, the axiom $H, L,$ return $va \xrightarrow{s} H, L,$ (Return,va,@empty) describes the completion type obtained when the return expression has fully evaluated to a pure value va.

4.2.3 Expression Semantics

We now describe the semantics of some of the key user and internal expressions. The key contextual rules for evaluating a sub-expression inside an outer expression and for propagating exceptions to the top level are the following

$$\frac{H, L, e \xrightarrow{e} H_1, L_1, e_1}{H, L, eC[e] \xrightarrow{e} H_1, L_1, eC[e_1]} \qquad H, L, eC[w] \xrightarrow{e} H, L, w$$

Here eC denotes an expression context for evaluating expressions. Examples of such contexts are _[e], va[_], typeof _, and so on (see [45] for the complete list).

Property Lookup. A property lookup is carried out using the internal expression @GetValue(r). If $r = l^*m$, then the evaluation of the expression involves looking up

property m of object at location l. On the other hand if r is a null reference then the evaluation throws an error.

$$\frac{Get(H,l,m) = va}{H, L, \texttt{@GetValue}(l\texttt{*}m) \xrightarrow{e} H, L, va} \qquad \frac{\begin{array}{l} o = new\_native\_error(\text{`` ''}, \#RefErrorProt) \\ H_1, l_1 = alloc(H, o) \end{array}}{H, L, \texttt{@GetValue}(\texttt{null*}m) \xrightarrow{e} H_1, L, \langle l_1 \rangle}$$

Here $Get(H,l,m)$ returns the value of property m of the object at location l. This function is recursively defined below

As discussed in Chapter 2, object property lookup in ES3 involves traversing the prototype-chain of the object. We formalize the prototype chain by having an internal property @Prototype in each object that stores a reference to its prototype object. The function $Get(H,l,m)$ is then recursively defined as follows.

$$\frac{m \notin H(l) \quad H(l).@Prototype = ln}{Get(H,l,m) = Get(H,ln,m)}$$

$$\frac{m \in H(l) \quad H(l).m = va}{Get(H,l,m) = va} \qquad Get(H, \texttt{@null}, m) = \texttt{@undefined}$$

In general, a reference appearing inside a larger expression is evaluated to a value by using the following contextual rule

$$H, L, eCgv(r) \xrightarrow{e} H, L, eCgv(\texttt{@GetValue}(r))$$

Here $eCgv$ denotes an expression context for evaluating references. Examples of such contexts are $\_[e]$, $va[\_]$, $v = \_$, ... (see [45] for a complete list). Each such context is also an expression context for evaluating expressions (eC).

Property Update. Property updates are carried out using the internal expression @PutValue(r,va). If $r = l\texttt{*}m$ then evaluation of the expression involves updating property m of object a location l with value va. Surprisingly, if $r = \texttt{@null*}m$ then the evaluation involves updating property m of the global object. The following rules makes this clear.

$$H, L, \texttt{@PutValue}(\texttt{null*}m,va) \xrightarrow{e} H, L, \#\texttt{Global.@Put}(m,va)$$
$$H, L, \texttt{@PutValue}(l\texttt{*}m,va) \xrightarrow{e} H, L, l.\texttt{@Put}(m,va)$$

Here l.@Put(m,va) is a special internal expression for carrying out the actual property update, whose semantics we define next. The semantics makes use of the predicate $CanPut(H, l, m)$ which checks if the first occurrence of property m on the prototype chain of object at location l does not have the attribute readOnly.

$$\frac{CanPut(H,l,m) \quad m \notin H(l) \quad H(l.m \leftarrow va) = H_1}{H, L, l.@Put(m,\ va) \xrightarrow{e} H_1, L, va} \qquad \frac{CanPut(H,l,m) \quad H(l).m = va_1\{[\tilde{a}]\} \quad H(l.m \leftarrow va\{[\tilde{a}]\}) = H_1}{H, L, l.@Put(m,va) \xrightarrow{e} H_1, L, va}$$

The above rules show that fresh properties are added with an empty set of attributes, whereas existing properties are replaced maintaining the same set of attributes

Identifier Lookup. An identifier x is resolved by traversing down the stack and looking for a scope object that has a property named "x", either directly or indirectly via inheritance. If such a scope object is found then a reference type consisting of the object location and the property name "x" is returned. Otherwise a @null reference is returned.

$$\frac{Scope(H, L, \text{"}x\text{"}) = ln}{H, L, x \xrightarrow{e} H, L, ln*\text{"}x\text{"}}$$

Here $Scope(H, L, \text{"}x\text{"})$ returns the location of the first (from the top of the stack) scope object that has a property named "x". It is recursively defined below. It makes use of the predicate $HasProperty(H, l, m)$ which checks if object $H(l)$ has a property m either directly or via inheritance.

$$Scope(H, emp, m) = @null$$

$$\frac{HasProperty(H,l,m)}{Scope(H, L{:}l, m) = l} \qquad \frac{\neg HasProperty(H,l,m)}{Scope(H, L{:}l, m) = Scope(H, L, m)}$$

Implicit Type Conversions. An important use of types is to convert the operands of typed operations and throw exceptions when the conversion fails. There are implicit conversions between strings, booleans, number and objects, and some of them can lead to the execution of arbitrary code. As an example, the evaluation of the member selection expression $l[va]$ involves converting the pure value va to a string. This is

handled by the following contextual rule

$$\frac{Type(va) \neq String \quad ToString(va) = e}{H, L, eCts[va] \xrightarrow{e} H, L, eCts[e]}$$

Here *eCts* represents expression contexts for string conversion, which includes the context l[_]. Each *eCts* context is also an expression context for evaluating references (*eCgv*) and therefore also an expression context for evaluating expressions (*eC*). The function *ToString* when applied to primitive values performs a straightforward side-effect free string coercion. For example, *ToString*(1) = "1", *ToString*(true) = "true" and so on.

The interesting case is that of location values (l), where *ToString*(l) is the internal expression l.@DefaultValue(String). The evaluation of this expression involves invoking the "toString", and possibly the "valueOf" methods of the object at location l. We explain the informal semantics here and leave the formal reduction rules to [45]. The first step is to invoke the "toString"' method of the object at location l. If the return value from this call is a primitive value then it is converted to a string, using the function *ToString*, and the evaluation terminates. If this value is not a primitive value or if the "toString" method does not exist, then the "valueOf" method is invoked. If the result from this invocation is not a primitive value or if the "valueOf" method does not exist then a TypeError exception is thrown. The following example illustrates this behavior:

```
ES3> var o={a:0};
ES3> o[{toString: function() {o.a = o.a+ 1; return "a"}}] % returns 1.
```

Assignment. As an example of how the semantics of user expressions are formalized, we present in detail the case of assignment expressions. The ES3 specification says that in order to evaluate an assignment expression: `LeftHandSideExpression = AssignmentExpression`, one needs to do the following:

```
1. Evaluate LeftHandSideExpression.
2. Evaluate AssignmentExpression.
3. Call GetValue(Result(2)).
```

```
4. Call PutValue(Result(1), Result(3)).
5. Return Result(3).
```

In our formalization, Steps 1, 2 and 3 are simply captured by contextual rules. In particular, Step 1 is obtained by making the context $\_ = e$ an expression context for evaluating expressions (eC). The contextual rule for eC contexts evaluate the inner expression to a final value v. Steps 2 and 3 are obtained by making the context $v = \_$ an expression context for evaluating references and therefore automatically an expression context for evaluating expressions. Steps 4 and 5 are captured by the following transition axiom:

$$H, L, v = va \xrightarrow{e} H, L, \texttt{@PutValue}(v,va)$$

The contextual rules for propagating exceptions take care of any exceptions raised during intermediate steps.

As a peculiar feature of the semantics, note that the evaluation of the expression $\texttt{x} = 42$, when identifier x is not present on the current stack, would amount to updating property "x" of the global object. This is because in this case the expression x would evaluate to a null reference and evaluating @PutValue on null references updates the global object.

Object Literal. The semantics of the object literal expression $\{pn_1{:}e_1,\ldots,pn_m{:}e_m\}$ makes use of three internal constructs @AddProp1, @AddProp2, and @AddProp3 to populate a newly created object with properties $ToString(pn_1), \ldots, ToString(pn_m)$, pointing to the result of expressions $e_1, \ldots, e_m$ respectively.

$$H, L, \{[\tilde{pn}{:}e]\} \xrightarrow{e} H, L, \texttt{@AddProp1}(\#\texttt{Object.@Construct}(),\{[\tilde{pn}{:}e]\}) \quad (1)$$

$$H, L, \texttt{@AddProp1}(l,\{pn_1{:}e_1,[\tilde{pn}{:}e]\}) \xrightarrow{e} H, L, \texttt{@AddProp2}(pn_1,e_1,l,\{[\tilde{pn}{:}e]\}) \quad (2)$$

$$H, L, \texttt{@AddProp2}(m,va,l,\{[\tilde{pn}{:}e]\}) \xrightarrow{e} H, L, \texttt{@AddProp3}(\texttt{@PutValue}(l\texttt{*}m,va),l,\{[\tilde{pn}{:}e]\}) \quad (3)$$

$$H, L, \texttt{@AddProp3}(v,l,\{[\tilde{pn}{:}e]\}) \xrightarrow{e} H, L, \texttt{@AddProp1}(l,\{[\tilde{pn}{:}e]\}) \quad (4)$$

$$H, L, \texttt{@AddProp1}(l,\{\}) \xrightarrow{e} H, L, l \quad (5)$$

Rule 1 (with help from the contextual rules) creates a new empty object, and passes control to @AddProp1. Rule 2 extracts the first property-expression pair (if it exists) from the object literal and passes control to @AddProp2. Next, contextual rules ensure that the property name gets converted to a string m and the expression gets evaluated to a value va. Rule 3 then creates a @PutValue expression for updating the property m of the object with value va. Once the @PutValue expression completes evaluation (as a result of a contextual rule), Rule 4 transfers control back to @AddProp1. Finally, when all property-expressions pairs have been extracted out, Rule 5 returns the reference to the updated object to the top level.

We deviate from the specification slightly in the way we create a new object (in Rule 1). The specification ambiguously states that a new empty object is created "as if by calling" new Object(). As suggested by various researchers, this is actually a bug in the semantics and the new object creation step should not execute new Object() explicitly. Instead a call to the "original" Object constructor should be *hard-wired* in the semantics. This is because in the case of an explicit call to new Object(), if the Object constructor is redefined or deleted by (malicious) code then the whole object literal mechanism might fail. Thus in order to make the mechanism more robust, we hardwire the call to the original Object constructor using the expression #Object.@Construct().

Binary logical operators. One may expect binary logical operators to have a dull semantics. That is not the case in ES3, as shown in the example below.

```
ES3> var a = true; if (a*1) 1 else 0 % result: 1.
ES3> var b = {valueOf: function() {return 0; }}; a = b&&b;
ES3> if(a*1) 1 else 0 % result: 0.
ES3> if (b) 1; else 0 % result: 1.
```

From the semantics it is immediately apparent that the && operator does not return a boolean value, but instead its second argument if the first one evaluates to true, or else its first argument. Combining this with implicit type conversions can be very confusing for a programmer. The formal semantics are described below. Both && and || expressions are translated to the same internal expression @L, modulo a flag

that is used (in xor with the result of converting the first parameter to a boolean) to determine which parameter to return. Let $\oplus$ be the xor operator. The evaluation rules are as follows

$$H, L, va \ \&\& \ \ e \xrightarrow{e} H, L, \texttt{@L(true,}va,\ va,\ e\texttt{)}$$
$$H, L, va \ || \ \ e \xrightarrow{e} H, l, \texttt{@L(false,}va,\ va,\ e\texttt{)}$$

$$\frac{b_1 \oplus b_2}{H, L, \texttt{@L(}b_1,\ b_2,\ va,\ e\texttt{)} \xrightarrow{e} H, L, va} \qquad \frac{\neg(b_1 \oplus b_2)}{H, L, \texttt{@L(}b_1,\ b_2,\ va,\ e\texttt{)} \xrightarrow{e} H, L, \texttt{@GV(}e\texttt{)}}$$

Equality checks. In ES3, there are two kinds of equality testing: strict equality which is without implicit type conversions and loose equality which is with implicit type conversions. Inequalities are defined as evaluating the logical negation of the corresponding equality.

$$H, L, e_1 =\neq= e_2 \xrightarrow{e} H, L, \neg(e1 === e2)$$
$$H, l, e1 =\neq e2 \xrightarrow{e} \neg(e1 == e2)$$

Strict equality can be implemented as loose equality modulo type equality:

$$\frac{Type(va_1) = Type(va_2)}{H, L, va_1 === va_2 \xrightarrow{e} H, L, va_1 == va_2} \qquad \frac{Type(va_1) \neq Type(va_2)}{H, L, va_1 === va_2 \xrightarrow{e} H, L, \texttt{false}}$$

Loose equality requires a lot of different rules to account for the different conversions required by the operands. We give only a few assorted cases, as an example.

$$H, L, \texttt{@null==@undefined} \xrightarrow{e} H, L, \texttt{true}$$

$$\frac{Type(va_1) = \texttt{Number} \quad Type(va_2) = \texttt{String}}{H, L, va_1 == va_2 \xrightarrow{e} H, L, va_1 == \texttt{@TN(}va_2\texttt{)}}$$

$$\frac{Type(va_1) = \texttt{Null} \quad Type(va_2) \neq \texttt{Undefined} \quad Type(va1) \neq Type(va2)}{H, l, va1 == va2 \xrightarrow{e} H, l, \texttt{false}}$$

Here @TN is a special internal expression for converting non-numeric value to numeric values.

4.2.4 Statement Semantics

We now describe the semantics of certain key statements. Recall from Section 4.2.2 that the evaluation of a statement terminates with a special completion value. The key contextual rule for evaluating a sub-statement of a large outer statement is the following

$$\frac{H, L, s \xrightarrow{s} H_1, L_1, s_1}{H, L, sC[s] \xrightarrow{s} H_1, L_1, sC[s_1]}$$

Here sC is a statement context for evaluating statements. Examples of such contexts include $\mathsf{try}\{\_\}\ \mathsf{catch(x)}\{s_1\}$, $\mathsf{@with(\_)}$, $\mathsf{@Blk2}(co,\_,\{s^*\})$, and so on (see [45] for the complete list).

Block Statements. A block statement $\{s^*\}$ in ES3 is essentially a sequential composition of a set of statements. Following the standard semantics of sequential composition, each statement in the block is executed in order and evaluation terminates as soon a statement evaluates to an abrupt completion. The abrupt completion is then propagated to the top level. Perhaps surprisingly, if all statements terminate normally then the final completion is the same as the completion obtained from the last statement, except that the value is the last non-empty value across the completions of all statements. The formal reduction rules are described below. They makes use of the internal statement $\mathsf{@Blk}$ and the function *Join* that takes two values vae_1, vae_2 as inputs and returns vae_1 if $vae_2 = \mathsf{@empty}$ and vae_2 otherwise. The context $\mathsf{@Blk}(co,\_,\{s^*\})$ is a statement context for evaluating statements.

$$H, L, \{\} \xrightarrow{s} H, L, (\mathsf{Normal,@empty,@empty}) \tag{1}$$

$$H, L, \{s_1[\ s^*]\} \xrightarrow{s} H, L, \mathsf{@Blk}((\mathsf{Normal,@empty,@empty}), s_1, \{s^*\} \tag{2}$$

$$\frac{Join(vae, vae_1) = vae \qquad ct_1 = \mathsf{Normal}}{H, L, \mathsf{@Blk}((ct,vae,xe),(ct_1,vae_1,xe_1),\{s_1[s^*]\}) \xrightarrow{s} H, L, \mathsf{@Blk}((ct_1,vae,xe_1),s_1,\{s^*\})} \tag{3}$$

$$\frac{Join(vae, vae_1) = vae \qquad ct_1 = \mathsf{Normal}}{H, L, \mathsf{@Blk}((ct,vae,xe),(ct_1,vae_1,xe_1),\{\}) \xrightarrow{s} H, L, (ct_1,vae,xe_1)} \tag{4}$$

$$\frac{ct \neq \mathsf{Normal}}{H, L, \mathsf{@Blk}(co,(ct,vae,xe),\{s_1[\ s^*]\}) \xrightarrow{s} H, L, (ct,\ vae,\ xe)} \tag{5}$$

With. The with statement is notoriously controversial. Its semantics is easy to explain on a technical level but it is hard to have the correct intuition while writing code using with.

The first step in the evaluation of with(e) s is to evaluate the expression e and convert the result to an object. This step is carried out by the contextual rule for expression evaluation within a statement, and the contextual rule for object conversion, which is defined below

$$\frac{Type(va) \neq Object \qquad ToObject(H, v) = H_1, l}{H, L, sCto[v] \xrightarrow{s} H, L, sCto[l]}$$

Here $sCto$ is any statement context for object conversion which includes the context with(_) s. The function $ToObject(H, v)$ returns a fresh number, string or boolean object (created by invoking the corresponding built-in constructors) if v is a number, string or boolean value respectively; it returns v if v is an object reference; otherwise it returns a fresh error object.

The second step is to place the object obtained from the first step at the top of the stack and then evaluate the statement s. When the statement completes evaluation, the stack is popped once. The following reduction rules make this clear. They make use of the internal statement @with, with the context @with(_) being a statement context for evaluating statements.

$$H, L, \text{with}(l)\ s \xrightarrow{s} H, L{:}l, \text{@with}(s) \qquad H, L{:}l, \text{@with}(co) \xrightarrow{s} H, L, co$$

In practice with(o){x = e} is equivalent to if (o.hasOwnProperty(x)) o.x = e; else x=e;, meaning that an attempt to update a non-existing property of o creates a global variable with the same name, rather than adding a property to o. This semantics composed with the semantics of the variable statement creates surprising behavior.

```
ES3> var n = {m:0}; with (n) {var m = 1}; n.m % result: 1.
ES3> m === undefined % result: true.
```

Above, var m = 1 is parsed before executing the program. It initializes m to @undefined in the global scope. At run time, the statement var m = 1 is essentially equivalent to

the statement m = 1, that sets the property "m" of object n to 1. The only difference is that var m=1 evaluates to a completion with value @empty, whereas m = 1 evaluates to a completion with value 1.

Try-catch. JavaScript provides a try-catch-finally mechanism to handle native and user-generated exceptions. We focus on the rule that does the interesting work, and that may raise some eyebrows.

$$\frac{\begin{array}{l} o = new\_object(\text{"Object"}, \#\text{ObjectProt}) \\ H_1, l_1 = alloc(H, o) \\ H_2 = H_1(l_1.\text{"x"} = va\{\text{DontDelete}\}) \end{array}}{H, L, \text{try (Throw,}va\text{, } xe\text{) catch (x) } \{s\} \xrightarrow{s} H_2, L{:}l_1, \text{@catch } \{s_1\}}$$

If the evaluation of the tried statement results in an exception, the catch code is executed in a new scope containing a binding of the catch variable x to the exception value va. Combined with the semantics of methods by self-application, comes the surprising ability of JavaScript programs of tamper with their own scope:

ES3> x=0; function spy() {return this};
ES3> try {throw spy} catch(spy) {spy().x = 1; x === 1} *% result: true.*
ES3> x *% result: 0.*

4.2.5 Program Semantics

Programs, as defined in Figure 4.3 are sequences of statements and function declarations. The first step in the evaluation of programs is the pre-processing of all function and variable declarations appearing in the program. This step involves initializing the current scope object with all function and variable declarations. We define functions $VD(H, l, \{\tilde{a}\}, P)$ and $FD(H, l, \{\tilde{a}\}, P)$ that take as arguments: heap H, current scope l, program P and attribute set $\{\tilde{a}\}$, and return the final heap obtained after initializing the current scope with all variable and function declaration respectively. The attribute set $\{\tilde{a}\}$ is used while adding properties to the current scope object. Evaluation of a program can be performed as part of a function call, as part of an eval

call or as part of top-level code. In the case of function call and top-level code, the attribute set used is {DontDelete}, whereas in the case of an eval call, the attribute set is empty {}.

The formal reduction rules for programs are defined below. They make use of two internal programs @Prog and @EvalProg.

$$H, L, P \xrightarrow{P} H, L, \text{@Prog}(P, \{\text{DontDelete}\}) \tag{1}$$

$$\frac{\begin{array}{c} VD(H, l, \{\text{DontDelete}\}, P) = H_1 \\ FD(H_1, l, \{\text{DontDelete}\}, P) = H_2 \end{array}}{H, L : l, \text{@Prog}(P, \tilde{a}) \xrightarrow{P} H, L : l, \text{@EvalProg}(P)} \tag{2}$$

$$\frac{ct \in \{\text{Normal}, \text{Return}\}}{H, L, \text{@EvalProg}((ct, vae, xe)\ P) \xrightarrow{P} H, L, \text{@EvalProg}(P)} \tag{3}$$

$$\frac{\begin{array}{c} ct \notin \{\text{Normal}, \text{Return}, \text{Throw}\} \\ o = new\_SyntaxError() \\ H_1, l_1 = alloc(H, o) \end{array}}{H, L, \text{@EvalProg}((ct, vae, xe)\ P) \xrightarrow{P} H, L, (\text{Throw}, l_1, \text{@empty})} \tag{4}$$

$$H, L, \text{@EvalProg}((\text{Throw}, vae, xe)\ P) \xrightarrow{P} H, L, (\text{Throw}, vae, xe) \tag{5}$$

$$H, L, \text{@EvalProg}(fd,\ P) \xrightarrow{P} H, L, \text{@EvalProg}((\text{Normal},\ \text{@empty},\ \text{@empty}),\ P) \tag{6}$$

Rule 2 initializes the current scope with all the variable and function declarations and passes control to @EvalProg which is responsible for evaluating the program. As usual the contextual rule for statements takes care of statement evaluation (@EvalProg(_ P) is a program context for evaluating statements). If the evaluation of a statement terminates with a completion with type Normal or Return, then evaluation proceeds to the next statement or function declaration in the sequence (Rule 3), otherwise evaluation is halted (Rules 4 and 5). Function declarations encountered during evaluation are ignored (Rule 6) as they have already been accounted for in the pre-processing step (Rule 2).

The evaluations of all top-level programs begin in a special initial heap H_0^{ES3}. The heap H_0^{ES3} is pre-populated with all the built-in objects, which are defined next.

4.2.6 Built-in Objects

The initial heap H_0^{ES3} of ES3 contains built-in objects for representing predefined functions, constructors, prototypes, and the global object @Global that constitutes the initial scope and is always the root of the scope chain. As an example, we describe the global object. The global object defines properties to store special values such as @NaN, @undefined, etc., methods such as "eval", "toString", etc., and constructors that can be used to build generic objects, functions, numbers, booleans and arrays. The heap location $\#global$ of the global object forms the base of every stack. Its @this property points to itself: @Global = { @this:#Global, "eval":#GEval{DontEnum},... }. None of the non-internal properties are read-only or enumerable, and most of them can be deleted. By contrast, when a user variable or function is defined in the top level scope (i.e. the global object) it always has the DontDelete attribute. The lack of a ReadOnly attribute on "NaN", "Number" properties forces programmers to use the expression 0/0 to denote the real @NaN value, even though @Number.NaN stores @NaN and is a read only property.

Eval. The @eval function takes a string and tries to parse it as a legal program text. If the parsing succeeds then the parsed program is evaluated, otherwise a SyntaxError is thrown. The evaluation rules are described below. $\#GEval$ is the heap location of the @eval function object. A call to the built-in eval function translates into the internal expression #GEval.@Exe(l_1,m) where l_1 is the this argument and m is the result of evaluating the string argument. The rules make use of the internal expression @cEval whose main purpose is to convert the completion obtained on evaluating the program to a value (see Rule 4 and 5). Program evaluation is handled by the internal program @Prog (see Rules 2 and 3). As we are not interested in modeling the parsing phase, we just assume a parsing function $ParseProg(m)$, which given a string m returns a

valid program P or else @undefined.

$$\frac{ParseProg(m) = \text{@undefined} \quad o = new\_SyntaxError() \quad H_2, l_2 = alloc(H, o)}{H, L, \#\text{GEval.@Exe}(l_1, m) \xrightarrow{P} H_2, L, \langle l_2 \rangle} \tag{1}$$

$$\frac{ParseProg(m) = P}{H, \#\text{Global}, \#\text{GEval.@Exe}(l_1, m) \xrightarrow{P} H_2, \#\text{Global}, \text{@cEval}(\text{@Prog}(P, \{\text{DontDelete}\}))} \tag{2}$$

$$\frac{l \neq \#\text{Global} \quad ParseProg(m) = P}{H, L{:}l, \#\text{GEval.@Exe}(l_1, \text{m}) \xrightarrow{P} H_2, L{:}l, \text{@cEval}(\text{@Prog}(\text{P}, \{\}))} \tag{3}$$

$$\frac{ct \neq \text{Normal}}{H, L, \text{@cEval}((ct, vae, xe)) \xrightarrow{P} H, L, \langle vae \rangle} \tag{4}$$

$$\frac{ct = \text{Normal} \quad Join(\text{@undefined}, vae) = v}{H, L, \text{@cEval}((ct, vae, xe)) \xrightarrow{P} H, L, v} \tag{5}$$

Object. The @Object constructor is a function object stored in the "Object" property of the global object. It is used directly for creating new user objects and also implicitly during the evaluation of object literals. Its "prototype" property points to the object @ObjectProt, which becomes the prototype of any object constructed using the @Object constructor. Invoked as a function or as a constructor, @Object returns its argument if it is an object, a new empty object if its argument is @undefined or not supplied, or converts its argument to an object if it is a string, a number or a boolean. If the argument is a host object (such as a DOM object) the behavior is implementation dependent. The following rules make this clear. Below, #Object is the heap location of the @Object constructor and #Object.@Construct(va) represents a constructor-call with argument va.

$$\frac{o = new\_object(\text{"Object"}, \#\text{ObjectProt}) \quad H_1, l_1 = Alloc(H, o) \quad Type(pv) \in \{Null, Undefined\}}{H, L, \#\text{Object.@Construct}(pv) \xrightarrow{P} H_1, L, l_1}$$

$$\frac{Type(l) = Object \quad \neg IsHost(H, l)}{H, L, \#\mathsf{Object.@Construct}(l) \xrightarrow{P} H, L, l}$$

$$\frac{Type(pv) \in \{String, Boolean, Number\} \quad H_1, l_1 = ToObject(H, pv)}{H, L, \#\mathsf{Object.@Construct}(pv) \xrightarrow{P} H_1, L, l_1}$$

The object @ObjectProt is the root of every prototype chain. For that reason, its internal prototype is @null. Apart from "constructor", which stores a pointer to @Object, its other public properties are built-in functions.

Function. The @Function constructor is a function object stored in the "Function" property of the global object. It is also reachable from the "constructor" property of any newly created function object. It takes a list of string names and a function body as input, and creates a function object with formal parameters as the names provided, and body as the function body provided. For example:

ES3> var f = Function(''x'', ''return x''); *% creates the function function(x){return x}.*

The Function constructor tries to parse its list of parameters into a (possibly empty) comma-separated list of formal parameters, and a program text. If the parsing succeeds, it returns a pointer to a newly allocated function.

$$\frac{H, \mathsf{Function(fun()\{;\},\#Global)} = H_1, l_1}{H, L, \#\mathsf{Function.@Construct()} \xrightarrow{e} H_1, L, l_1}$$

$$H, L, \#\mathsf{Function.@Construct}([v\tilde{a}_1,]va_2) \xrightarrow{e} H, L, \mathsf{@FunParse}(\text{`` ''},[v\tilde{a}_1,]va_2)$$

We skip the reduction rules for the @FunParse expression which can be found in [45]. Oddly, instead of its proper lexical scope, the created function gets the global object as its scope, by the expression $H, \mathsf{Function(fun(\tilde{x})\{P\},\#Global)} = H_1, l_1$. Hence, the following behavior arises

ES3> (function () {var b=1; var c= function () {x=b}; c()})(); x *% result: 1.*
ES3> (function () {var b=1; var c= new Function(''x=b''); c()})(); x
% error: ReferenceError b is not defined.

For this reason, the @Function constructor is rarely used by programmers and functions are usually created by function declarations or function expressions. However, regardless of how a function object is created, its prototype is always the object @FunctionProt which is pointed to by the "prototype" property of @Function.

The object @FunctionProt is a function itself, which just returns @undefined and defines fields such as

$$\text{@FunctionProt} = \left\{ \begin{array}{l} \text{@Class: Function,} \\ \text{@Prototype: \#ObjectProt,} \\ \text{constructor: \#Function\{DontEnum\},} \\ \text{"prototype": \#FunctionProtProt\{DontDelete\},} \\ \text{"call": \#FPcall\{DontEnum\},} \\ \ldots \end{array} \right\}$$

$$H, l, \#\text{FunctionProt.@Exe}(l_1) \xrightarrow{e} H, l, \text{@undefined}$$

Its method "call" is very characteristic of ES3, exposing the very essence of the language to the programmer. ES3, at the core, is a functional language where objects are records, and where functions are first-class values. The typical behavior of object-oriented languages is achieved by parameterizing functions by an implicit first argument which represents the object/record of which they are a method. The method invocation o.f(v) is essentially equivalent to (o.f).call(o,v), where first we extract function f from object o, and then we use the "call" method to apply f to o and its argument v.

By de-sugaring method invocation, the "call" method makes it possible to turn arbitrary functions into methods without modifying the target object. For example, we can use the standard "toString" method of object on arrays, which have their own different "toString".

```
ES3> var a = [1,2,3]; a.toString() % result: 1, 2, 3.
ES3> toString.call(a) % result: [object Array].
```

We refer to the reader to [45] for the complete formal semantics of the "call" method.

4.3 Relation to Implementations

So far we have referred to ES3 as the language defined by the 3^{rd} edition of the ECMA-262 specification. ES3 is also implemented by all major web browsers, and by standalone shells. In order to clarify several ambiguities present in the specification, we ran experiments inspired by our semantics rules on different implementations. We found that, besides resolving ambiguities in different and often incompatible ways, implementations sometimes do not conform to the specification in implementing corner (and not so corner) cases which have a well-defined semantics. In this section we describe, mainly by examples, some interesting differences and incompatibilities between ES3 implementations. Note that we do not cover language extensions (such as *getters* and *setters*) developed by the various implementations on top of the standard language. The implementations that we considered were current between March to December 2008, which was the period during which this research was carried out. In our discussion we use the prompt SM> to denote the Mozilla SpiderMonkey interpreter (release 1.7.0). RH> to denote the Rhino interpreter (release 1.7R1) and MS> to denote Microsoft's JScript interpreter (release 7.0). The prompt ES3>, as before, represents a hypothetical strictly ECMA-262, 3^{rd} edition compliant interpreter.

Function expressions and declarations. This is one of the most striking examples of implementations diverging from the specification. As prescribed by the specification, a term cannot be parsed as a statement if it starts with function. Hence s_1; function a(){}; s_2 can be parsed as a program but cannot be parsed as part of a block statement. Function definitions, according to the specification, are parsed in source text order before running the program code, and are added to the scope. Hence, the code below is fully specification compliant in choosing the second function declaration in the first example, and throwing an exception in the second example.

```
ES3> (function () {
       function b() {return 1;}
       return b();
       function b() {return 0;}
     })() % result: 0.
```

```
ES3> (function () {function b() {return 1;}}; return b())() % result: SyntaxError...
```

Major implementations instead accept function definitions as statements, with mixed results. From our experiments, we think that Rhino parses standard-compliant function definitions according to the specification, removes them from the parsed code, but considers at run time (with the same semantics) the remaining function definitions that appear as statements.

```
RH> (function () {
      function b() {return 0};
      if (true)
        function b() {return 1;}
      return b();})() % result: 1.
RH> (function () {
      function b() {return 0;}
      if (false)
        function b() {return 1;}
      return b()})() % result: 0.
```

We find this behavior completely reasonable, and easy to implement in a formal semantics. Our experiments on SpiderMonkey gave different results. Apparently, the semantics of functions in this implementation depends on the position, *within unreachable code* of statements (such as var g below) which should have no semantic significance!

```
SM> (function() {
      if (true)
        function g() {return 0;}
      return g();
      var g;
      function g() {return 1;}})() % result: 0.
SM> (function() {
      if (true)
        function g() {return 0};
```

```
    return g();
    function g() {return 1;};
    var g})() % result: 1.
```

JScript parses all the functions declarations at the beginning, including the ones appearing as statements.

```
MS> if(true) {
      function a() {return 1;}
    } else {
      function a() {return 0;}
    };
    a(); % result: 0.
```

Internal creation of new objects. We have shown in Section 4.2.3 that literal objects are created by evaluating the expression new Object(), instead of creating internally a new empty object as described in the specification. The specification is ambiguous regarding this and several similar cases. We followed the main implementations in choosing where to evaluate the expression new Object and when to use a fresh empty object as described in the specification. For example, the latter option is popular for creating the scope object in the catch branch of a try-catch construct.

Native functions used as constructors. The specification dictates that native functions cannot be used as constructors unless explicitly mentioned in their description. We reflected this in our semantics by not providing native functions (such as eval) with a @Construct method, but the SpiderMonkey implementation has chosen not to do so.

```
SM> var a = new eval()
SM> a % result: [object Object].
SM> var a = new Function.prototype()
SM> a % result: [object Object].
```

Function arguments aliasing. The actual parameters of a function call are collected in an arguments array-like object available inside the function scope. The fields

of this array are aliases for the formal parameters:

```
ES3> (function (x) {arguments[0]=1; return x})(4) % result: 1.
```

This is an odd feature, the only case where such aliasing is observable in ES3. This feature has now been removed from the standard. We decided to not model this feature in our semantics as it required extra machinery, and did not seem to affect the semantics of programs in an interesting way, apart perhaps from confusing the programmer. The SpiderMonkey implementation, however, seems to be implementing the aliasing in an unusual manner, perhaps using getters and setters. If the particular property of the arguments object is deleted then the aliasing is broken.

```
SM> (function(x) {delete arguments[0]; arguments[0]=1; return x})(4) % result: 4.
```

We do not believe that this behavior is entirely justified, as the specification can be interpreted as requiring that the aliasing should be in place whenever arguments has a property corresponding to the position of an existing formal parameter.

Joined objects. The specification provides for the possibility that functions defined by the same piece of source text be implemented as joined objects, i.e. sharing their properties. If that were the case, we could have the following behavior.

```
ES3> function f() {function g() {}; return g;}
     var h = f();
     var j = f();
     h.a = 0;
ES3> i.a % result: 0.
```

Fortunately, no known implementation uses this feature, and we do not model it in our semantics. This feature has been removed from future versions of the language.

Scoping of the catch construct. We have shown in Section 4.2.4 that the scoping mechanism of the try-catch construct can lead to programs getting hold of their own scope. SpiderMonkey and other implementations decided to protect the programmer from such abomination, and pass instead the global object as the implicit this parameter to the spy function below.

```
SM> x=0; function spy() {return this;}
```

```
SM> try {
      throw spy;
    } catch(spy) {
      spy().x = 1;
      x === 1;
    } % result: true.
JS> x % result: 1.
```

Our semantics respects the ES3 specification.

4.4 Analysis Framework

In this section we give some preliminary definitions and set up a basic framework for formal analysis of well-formed ES3 programs. We prove a progress and preservation theorem which shows that the semantics is sound and the execution of a well-formed term always progresses to an exception or an expected value. Although this property is fairly standard for idealized languages used in formal studies, proving it for real (and unruly!) ES3 is a much harder task. We begin by setting certain notation and definitions. The formal analysis framework that we develop below will also be used in Chapters 5 and 6 where we analyze sandboxing mechanisms for ES3.

4.4.1 Notation and Definitions

We define *Exprs*, *Stmts*, *Progs* respectively as the sets of all possible expressions, statements and programs that can be written using the corresponding internal and user-level grammars. $Terms^{ES3} := Exprs \cup Stmts \cup Progs$ is therefore the set of all possible ES3 terms that can appear in a state. $Terms^{ES3}$ is split as $Terms^{ES3} := Terms^{ES3}_{user} \uplus Terms^{ES3}_{int}$ where $Terms^{ES3}_{user}$ and $Terms^{ES3}_{int}$ are the sets of all user-level and internal terms respectively. Furthermore, we define *Vals* as the set of final values that can be obtained on evaluating a term. *Vals* is split as $Vals := NVals \uplus EVals$ where *NVals* is the set of normal values, defined as $\{va\} \cup \{ct \mid Type(ct) = \mathsf{Normal}\}$, and *EVals* is the set of error values, defined as $\{w\} \cup \{ct \mid Type(ct) \neq \mathsf{Normal}\}$. We

use *Locs*, *Props* as the universe of heap locations, and object properties respectively. *Props* is split as $Props := Props_{user} \uplus Props_{int}$ where $Props_{user}$ and $Props_{int}$ are the set of all possible user and internal properties respectively.

Recall from Section 4.2.2 that a heap is a map from heap locations to objects, an object is a record from properties to pure values or closures, and a stack is a list of heap locations. For a heap location l, a term t and a stack L, we say $l \in t$ iff l occurs in t and $l \in L$ iff l is present on the stack L. Given a heap H, we define $dom(H)$ as the set of all allocated heap locations of H, and given an object o, we define $props(o)$ as the set of all properties of o. We split $props(o)$ as $props(o) := prop^{user}(o) \uplus prop^{int}(o)$ where $prop^{user}(o)$ and $prop^{int}(o)$ are the sets of all user and internal properties of object o. H_0^{ES3} is the initial heap containing all the built-in objects and $L_0^{ES3} := emp : \#global$ is the initial stack containing only the global object. $Builtins := dom(H_0^{ES3})$ denotes the set of heap locations of all the built-in objects.

Given a state S, we use $heap(S)$, $stack(S)$ and $term(S)$ to denote the heap, stack and term parts of a state. We use $\rightarrow$ as the union of the relations $\xrightarrow{e}$, $\xrightarrow{s}$ and $\xrightarrow{P}$. Given states S and T, we say S reaches T in many steps, denoted by $S \rightsquigarrow T$, iff either $S \rightarrow T$ holds or there exists states $S_1, \ldots, S_n$ $(n \geq 1)$ such that $S \rightarrow S_1 \rightarrow \ldots \rightarrow S_n \rightarrow T$ holds. For a state S, $\tau(S)$ is the possibly infinite sequence of states $S, S_1, \ldots, S_n, \ldots$ such that $S \rightarrow S_1 \rightarrow \ldots \rightarrow S_n \rightarrow \ldots$. We write $S \uparrow$ when $\tau(S)$ is non-terminating and $final(\tau(S))$ for the final state if trace $\tau(S)$ is finite. A state S is *initial* if $heap(S) = H_0^{ES3}$, $stack(S) = L_0^{ES3}$ and $term(S) \in Terms_{user}^{ES3}$.

Well-formedness. We define the notion of *well-formedness* for $ES3$ states. A $S = (H, l, t)$ is well formed iff the heap H is well formed, and the stack L and term t are well formed with respect to the heap H. We formalize the property as $Wf(S) := Wf_h(H) \wedge Wf_s(L, H) \wedge Wf_t(t, H)$. The definition of $Wf_h(H)$ for a heap H is given in Figure 4.6. It is essentially a big conjunction of a set of invariants that must hold for a heap. $Wf_s(L, H)$ holds for a stack L iff the bottommost element of L is #global and $\forall l : l \in L \Rightarrow l \in dom(H)$. Finally, $Wf_t(t, H)$ holds for a term t if iff $t \in Terms^{ES3}$ and $\forall l : l \in t \Rightarrow l \in dom(H)$.

Actions. From the semantics of ES3 it is clear that every heap action in ES3 is

$Wf_h(H)$ *holds for a heap* H *is iff the following conditions are true.*

- *Every allocated object has the internal properties* @class *and* @prototype

$$\forall l \in dom(H) : \{\text{@class}, \text{@prototype}\} \subset props(H(l))$$

- *Every allocated Function object has the properties* @call, @construct, "prototype", *and* "length", *and the stack and function value stored in the* @closure *property are well formed.*

$$\forall l \in dom(H(l) : H(l).\text{@class} = \text{"Function"} \Rightarrow \left(\begin{array}{l} \{\text{@call}, \text{@construct}, \text{"prototype"}, \text{"length"}\} \subseteq props(H(l) \\ \wedge\ \forall fv, L : H(l).\text{@closure} = fv, L \Rightarrow Wf_t(fv) \wedge Wf_s(L) \end{array} \right)$$

- *Every built-in function object has the property* @actuals.

$$\forall l \in Builtins : H(l).\text{@class} = \text{"Function"} \Rightarrow \text{@actuals} \in props(H(l)$$

- *Every allocated Array object has a* "length" *property whose value is always of type* Number.

$$\forall l \in dom(H(l) : H(l).\text{@class} = \text{"Array"} \Rightarrow Type(H(l).\text{"length"}) = \text{"Number"}$$

- *Every allocated String, Number and Boolean object has the* "@value" *property.*

$$\forall l \in dom(H) : H(l).\text{@class} \in \{\text{"String"}, \text{"Number"}, \text{"Boolean"}\} \Rightarrow \text{@value} \in props(H(l)$$

- *Every allocated arguments object has the properties* "callee" *and* "length"

$$\forall l \in dom(H) : H(l).\text{@class} = \text{"Arguments"} \Rightarrow \{\text{"callee"}, \text{"length"}\} \in props(H(l)$$

- *Every allocated error object has the message property.*

$$\forall l \in dom(H) : H(l).\text{@class} = \text{"Error"} \Rightarrow \text{"message"} \in props(H(l)$$

- *The values of all internal properties of built-in objects is the same as that on the initial heap* H_0^{ES3}.

$$\forall l \in Builtins, p : p \in prop^{int}(H_0^{ES3}(l)) \Rightarrow H(l).p = H_0^{ES3}(l).p$$

- *The prototype chain for any object never contains a cycle.*

Figure 4.6: Well-formedness of ES3 heaps

essentially a property read or write. Thus we abstractly specify a heap action by a triple of the form l, p, d where d is r or w, respectively indicating a *read* or *write* action to a property p of object at heap location l. Note that we do not model allocation of heap locations as heap actions as these actions are not relevant to the analysis we carry out in this work. We let $\mathbb{D} := \{\mathsf{r}, \mathsf{w}\}$ and define the universe of all possible actions by $Actions := Locs \times Props \times \mathbb{D}$. For a heap H, we define $Actions(H)$ as the set of all heap actions valid for the heap H. Formally,

$$Actions(H) := \{l, p, d \mid l \in dom(H) \ \wedge \ p \in Props_{user} \ \wedge d \in \mathbb{D}\}$$

Given a state S, we let $Act(S)$ as the set of all heap actions performed during a single step evaluation of a state S. For example, during the evaluation of the state H, L, x by one step, the set of actions performed are $\{(l_1, \mathsf{x}, \mathsf{r}), \ldots, (l_k, \mathsf{x}, \mathsf{r})\}$ where $l_1, \ldots, l_k$ are the top k locations on the stack L such that $\mathsf{x} \in Props(H(l_k))$ and $\mathsf{x} \notin Props(H(l_i))$ for $i = 1, \ldots k-1$. The map Act is naturally extended to a reduction trace τ by considering the union of all heap actions performed during every single-step reduction present on the trace τ. Finally for an action a, we abuse notation and use $loc(a)$, $props(a)$ and $perm(a)$ to denote the location, property name and permission part of the action respectively.

4.4.2 Progress and Preservation

We now present the main technical result which is a progress and preservation theorem showing that evaluation of a well-formed state never gets stuck (Progress) and that well-formedness of states is preserved across evaluation (Preservation).

Theorem 1 (Progress and Preservation) *For all well-formed states S_1:*

- *(Progress) If $term(S_1) \notin Vals$ then there exists a state S_2 such that $S_1 \rightarrow S_2$.*
- *(Preservation) If there exists a state S_2 such that $S_1 \rightarrow S_2$, then S_2 is well-formed.*

The proof of the above theorem is carried out using an induction on the set of reduction rules for the preservation part, and a structural induction on the terms for the

progress part. Due to the sheer volume of the semantics, we only describe a sketch of the proof in the appendix (Section A.1).

4.5 Related Work

The closest work on the semantics of JavaScript at the scale of the ECMA specification is [27]. It describes the semantics by defining a translation of JavaScript to a core calculus *LambdaJS* and then providing execution rules for expressions in LambdaJS. A huge upside of this approach is that the semantics of LambdaJS fits in three pages and wields itself very well to manual proofs. The calculus supports lexical scoping and is very well suited to developing type-based analyses for JavaScript as illustrated by further work [28, 68]. However proving properties about the surface JavaScript language requires the extra step of translating those properties to LambdaJS. Moreover, compliance with the ECMA specification or existing browser implementations relies critically on the translation. The authors do justify their translation by testing it using existing browser test suites, but this does not provide a formal guarantee. While proving a formal compliance guarantee is impossible for our semantics as well, our semantics is designed as a *transliteration* of the specification and therefore makes specification-compliance mostly apparent.

Other prior foundational studies of JavaScript include an operational semantics for a subset of JavaScript [30] and formal properties of subsets of JavaScript [5, 75, 84, 83]. [5] formalizes a small subset of JavaScript and give a static type system that prevents run-time typing errors. The subset is non-trivial, as it includes dynamic addition of properties to objects and constructor functions to create objects. However the subset also lacks important features such as object prototyping, functions as objects, statements such as with, try-catch, for-in, and native functions and objects. This leads to substantial simplifications in their semantics, relative to ours. [84] proposes a type system for a larger subset of JavaScript than [5], as it also includes function expressions, function objects, and object literals. The type system associates type signatures with objects and functions, and identifies suspicious type conversions. However, this subset still does not allow object prototyping, the with and the try-catch

statements, or subtle features of the language such as hoisting of variable declarations in the body of a function.

Our ultimate goal in this work has been to analyze the correctness of sandboxing mechanisms designed for untrusted ES3 code in the wild. This requires reasoning about all possible ES3 code contexts. As a result our semantics is substantially different from these previous efforts and addresses the full ECMA Standard language (with provisions for variants introduced in different browsers).

Chapter 5

Hosting Page Isolation

In this chapter[1], we design a language-based sandboxing mechanism for enforcing *Hosting page Isolation* on a single third-party component. The hosting page isolation property, introduced informally in Chapter 3, requires that third-party code must *only* access certain *white-listed* global variables. In accordance with the API+LBS architecture (see Chapter 3), the white-listed global variables are ones that hold references to the API objects. Similar to sandboxing mechanisms surveyed in Chapter 3, our mechanism is based on a combination of *filtering* and source-to-source *rewriting* of third-party code, and on *wrapping* certain objects in the hosting page's execution environment. Our analysis and security proofs are based on the semantics of ES3 [45], described in Chapter 4.

As part of this work, we compare our sandboxing mechanism to Facebook FBJS. Our analysis reveals multiple security vulnerabilities (reported in [50], [47]) in the FBJS enforcement of hosting page isolation. These vulnerabilities, which have now been fixed, allow a malicious application to read sensitive non-whitelisted global variables. On the expressiveness side, we find our mechanism to be as expressive as FBJS, from an application developer's viewpoint. The work presented in this chapter was carried out between June 2008 to April 2009, and the versions of FBJS analyzed are the ones that were current in Novement 2008 and March 2009.

Organization. The rest of this Chapter is organized as follows. Section 5.1 formally

[1]This chapter is based on joint work with Sergio Maffeis.

states the Hosting Page Isolation problem, and the challenges involved in solving the problem. Section 5.2 reviews the semantics of ES3 and identifies certain key facts that are important for designing sandboxing mechanisms. Section 5.3 then uses these facts to motivate and define a sandboxing mechanism for solving the hosting page isolation problem, and then states the main correctness theorem for the mechanism. Section 5.4 compares our mechanism with FBJS and presents the security vulnerabilities found in FBJS as part of the process. Finally Section 5.5 discusses related work.

5.1 Hosting Page Isolation Problem

In this section, we formally state the Hosting Page Isolation problem for ES3. In doing so we make use of the analysis framework developed in Section 4.4.1. We start by formally defining sandboxing mechanisms for third-party code.

5.1.1 Sandboxing Mechanism

As discussed in Chapter 3, a language-based sandboxing mechanism is a combination of filtering, rewriting and wrapping.

Definition 1 (Single-Component Sandboxing Mechanism) *A language-based sandboxing mechanism for a single third-party component is tuple $\langle H, L, \phi \rangle$ consisting of a heap H, stack L and a function $\phi : Terms^{ES3}_{user} \rightarrow Terms^{ES3}_{user}$. The mechanism is* well-formed *iff the heap H and stack L are well-formed, and for all terms $t \in Terms^{ES3}_{user}$, if t is well-formed with respect to the heap H then $\phi(t)$ is also well-formed with respect to the heap H.*

For a well-formed sandboxing mechanism $\langle H, L, \phi \rangle$, the heap H and stack L model the initial execution environment for third-party code, defined possibly by wrapping all security-critical resources. The function ϕ models the filtering and rewriting applied on third-party code. While modeling source-to-source rewriting by a function ϕ is straightforward, filtering is modeled by mapping all terms that do not pass through the filter, to an innocuous term, say `0`.

5.1.2 Problem Statement

In order to formally define the problem, we first define the Hosting Page Isolation property for third-party code. Informally the property states that third-party code only accesses global variables named in the *whitelist* $\mathcal{G}$. The formal definition of the property makes use of the map $Act(\tau)$ (see Chapter 4, Section 4.4.1) which for an evaluation trace τ, gives the set of all heap actions occurring during the trace. A heap action is denoted by a triple l, p, d consisting of a location l, property name p and permission d indicating whether property p of location l is read or written.

Definition 2 (Hosting Page Isolation) *Given a whitelist $\mathcal{G}$ of global variables, a third-party term $t \in Terms^{ES3}_{user}$ achieves hosting page isolation on a heap H and stack L, iff $Wf(H, L, t)$ implies*

$$\forall a : (a \in Act(\tau(H, L, t)) \ \wedge \ loc(a) = \#global) \implies props(a) \in \mathcal{G}.$$

This property is formally denoted by $Isolation^{h}_{\mathcal{G}}(H, L, t)$.

We now formally state the problem

> *Given a whitelist $\mathcal{G}$ of global variables, design a well-formed sandboxing mechanism $\langle H, L, \phi \rangle$ such that for all third-party terms t, the term $\phi(t)$ achieves hosting page isolation.*

5.1.3 Challenges and Approach

Hosting page isolation requires that third-party code only accesses whitelisted global variables. In most conventional lexically scoped languages, this is straightforward to check as a static scope analysis can determine the set of global variables accessed by a program. If the program accesses any global variable outside the whitelist then it can simply be discarded. Unfortunately in ES3, it is impossible to statically determine whether an identifier occurrence binds to a global variable! Consider the following example:

```
var critical = 42; % non-whitelisted global variable
```

```
% begin third-party code
var f = function() {
          var critical = 42;
          ...
          % some more code
          ...
          return critical;
        }
f();
% end third-party code
```

Let us imagine that the above code is run in an environment that has a non-whitelisted global variable critical. We ask the question: *does the call f() involve accessing the global variable critical?* In most lexically scoped languages, the answer is *No*, regardless of the code between the var declaration and return statement. However this is not true for ES3. If ⟨some code⟩ is replaced with x = this.x, then the answer is *Yes*. This is because in this case the this argument of the function f binds to the global object, and the property "critical" of the global object is the global variable critical. To complicate the analysis further, one could write x = this[e] where e is some difficult to analyze expression that evaluates to "critical" at runtime.

Even if third-party code is statically disallowed from using this, it can still access the global variable critical by replacing <some code> with eval("critical = this.critical"). In this case the code accessing the global variable critical only comes into existence at run-time and therefore detecting it is outside the scope of static analysis.

In summary, dynamic code generation and the fact that scope objects can be accessed as first-class objects make enforcing hosting page isolation very challenging for ES3. In order to overcome these challenges, we study the semantics of ES3 and systematically analyze all constructs that allow property/variable access and dynamic code generation (Section 5.2). We then design filtering and rewriting mechanisms to *tame* each of these constructs so that they respect the whitelisting policy (Section 5.3).

5.2 Design Principles

In this section we informally summarize some of the key insights that we gained while formalizing the operational semantics of ES3 [45]. We make use of these insights in Section 5.3 to formally design a sandboxing mechanism for enforcing hosting page isolation. We refer the reader to Chapter 4 for an overview of our semantics of ES3.

5.2.1 Property Access

We begin by enumerating all user-level language constructs whose evaluation involves reading or writing object properties. Certain properties get accessed implicitly during the evaluation of a term. For instance, the creation of an object literal involves accessing the property "Object" of the global object . Therefore we classify property accesses into two groups: *explicit* and *implicit*. We note that certain constructs, such as p in o, allow reflecting upon the properties of an object. We do not consider these constructs as they don't involve reading or writing the contents of an object property.

Explicit property access. We discuss user-level constructs which when evaluated involve accessing a property whose name is either present textually in the construct or is the result of evaluating a sub-expression present textually in the construct.

Fact 1 *There are only three kinds of expressions in ES3 which can be used for explicit property access:* x, $e.x$ *and* $e_1[e_2]$

Below, we discuss the semantics of each of the expressions x, $e.x$ and $e_1[e_2]$..

- Expression x: This is the Identifier expression whose evaluation involves accessing a property named "x". The semantics are described in detail in Section 5.3. The evaluation of the expression x involves successively looking at objects on the scope stack until an object with property "x" is found. Since object property lookup can also involve going up the prototype chain, the property "x" may also be accessed from an object on the prototype chain of a scope object.
- Expression $e.x$: This is the standard *dot* notation based property access mechanism whose evaluation involves accessing the property named "x" of the object

obtained by evaluating the expression e.

- Expression $e_1[e_2]$: This is the most unusual property access mechanism in ES3. It involves accessing the property name corresponding to the string form of the value obtained by evaluating the expression e_2. The actual evaluation of $e_1[e_2]$ goes through the following steps (informally): First e_1 is evaluated to a value va_1, and then e_2 is evaluated to a value va_2. Next, value va_1 is converted to an object o, followed by a converting value va_2 to a string m. Finally property m of object o is accessed. In summary the steps are: $e_1[e_2] \xrightarrow{e} va_1[e_2] \xrightarrow{e} va_1[va_2] \xrightarrow{e} o[va_2] \xrightarrow{e} o[m]$. Note that each of these steps may raise an exception or have other side effects.

Implicit property access. These take place when the property accessed is *not* named explicitly by the term, but is accessed as part of an intermediate evaluation step in the semantics. For example, the "toString" property is accessed implicitly by evaluating the expression "a" + o, which involves resolving the identifier o and then converting the value obtained to a string, by calling its "toString" property. There are many other expressions whose execution involves implicit property accesses to native properties, and the complete set is hard to characterize. Instead, we enumerate the set of all property names that can be implicitly accessed.

Fact 2 *The set of all property names $\mathcal{P}_{nat}$ that can be accesses implicitly by ES3 constructs is* $\{0,1,2,...\} \bigcup \{$*toString, valueOf, length, prototype, constructor, message, arguments, Object, Array, RegExp*$\}$

5.2.2 Dynamic Code Generation

ES3 also has constructs that allow dynamic code generation. For example, the built-in function @eval takes a string as an argument, parses it as a program, and evaluates the resulting program returning its final value. These constructs pose a challenge in designing filtering and rewriting-based sandboxing mechanisms for ES3, as they allow new code to appear dynamically. According to the operational semantics, in

ES3 there are only functions @eval and @Function that can dynamically generate new code. Thus we have the following fact.

Fact 3 *The only ES3 constructs which involve dynamic code generation (from strings to Programs) are the built-in functions @eval, @Function, which in the initial heap H_0 are only reachable via the properties "eval" and "Function" of the global object, and the property "constructor" of any function object.*

5.2.3 Accessing Global Variables

In ES3 global variables are essentially properties of the global object. They may get accessed implicitly during the execution of a term (see Fact 2), or during identifier lookup, or while reading properties directly off the global object.

Fact 4 *A global variable may be accessed implicitly during the execution of a term if the variable belongs to the set $\mathcal{P}_{nat}$, or during execution of an identifier expression x or expressions of the form $e_1.x$ and $e_1[e_2]$ where e_1 evaluates to a reference to the global object.*

Since controlling access to the global object is crucial in isolating global variables, we also explore the set of constructs that can be used to obtain a reference to the global object. As our semantics is formulated, the global object for the initial heap state is only accessible via the internal property @this. This property can in turn be accessed as a side effect of executing other instructions. An analysis of our semantics shows that the only construct whose evaluation involves access to the @this property is the expression this. Besides using this, the global object can be returned by calling in the global scope the functions "valueOf" of Object.prototype, and "concat", "sort" or "reverse" of Array.prototype. As an example, var f=Object.prototype.valueOf; f() evaluates to the global object.

Fact 5 *The only ES3 constructs that can return a reference to the global object are: the expression this, the built-in method "valueOf" of Object.prototype, and the built-in methods "concat", "sort" or "reverse" of Array.prototype.*

5.3 Sandboxing Mechanism

In this section we develop a sandboxing mechanism for enforcing hosting page isolation, using the insights gained from Section 5.2. We assume that we are provided a whitelist $\mathcal{G}$ of global variables. From fact 4, we know that global variables can either be accessed implicitly during the execution of a term, or explicitly using an identifier expression, or by reading properties of the global object. While the explicit property accesses can be controlled by appropriately filtering and rewriting third-party code, implicit property accesses are very difficult to control. In particular, given a third-party term, it is undecidable to statically decide the precise list of property names that will be accessed implicitly. On the positive side however, from Fact 2, we know that all implicitly accessed properties are contained in the set $\mathcal{P}_{nat}$. In this work we only focus on enforcing whitelists $\mathcal{G}$ that include $\mathcal{P}_{nat}$.

Assumption 1 $\mathcal{P}_{nat} \subseteq \mathcal{G}$.

We note that the only objects reachable on the initial heap from global variables named in $\mathcal{P}_{nat}$ are the functions stored int the properties "toString" and "valueOf" of Object.prototype and the built-in @Object, @Array and @RegExp constructors. Since these objects provide essential functionality to code, it is reasonable to assume that in most applications third-party code would be permitted to access them and the variables named in $\mathcal{P}_{nat}$ would be whitelisted.

In the remaining part of this section we develop various filters and rewritings that prevent untrusted code from the following: (1) using identifiers that resolve to non-whitelisted global variables, (2) obtaining a direct reference to the global object, and (3) carrying out dynamic code generation. The third restriction is necessary as dynamic code generation provides a means for bypassing the filtering and rewriting techniques developed for (1) and (2).

5.3.1 Restricting Identifiers

The first step is to restrict third-party code from using any identifier name that resolves to a non-whitelisted global variable. As we saw in Section 5.1.3, statically

determining wether an identifier resolves to a global variable is undecidable in ES3. Thus we conservatively disallow *all* identifiers that are not named in the whitelist $\mathcal{G}$. This restriction is modeled using the following filter

Filter 1 *Disallow all terms which contain an identifier x such that "x" $\notin \mathcal{G}$.*

5.3.2 Isolating the Global Object

From Fact 5 we know that a reference to the global object can be obtained only via the expression this, method "valueOf" of Object.prototype and methods "sort", "reverse", "concat" of Array.prototype. We use rewriting to restrict the behavior of this and wrapping for restricting the behavior of the aforementioned built-in methods.

Rewriting this. The main idea is to rewrite every occurrence of this in the user code to the expression NOGLOBALTHIS which behaves as this, if this does not evaluate to the global object, and null otherwise. In order to define NOGLOBALTHIS and other such rewriting mechanisms, we need references to certain built-in objects such as the global object, the built-in @String constructor, etc. Since all properties of built-in objects can be dynamically modified by third-party code, it is important to store references to these objects in a certain trusted state that is inaccessible to third-party code. In this work, we use '$' prefixed properties of Object.prototype as the trusted state, and explicitly forbid third-party code from accessing any '$' prefixed property of any object. Trusted expressions such as NOGLOBALTHIS access this state by creating a new empty object (that automatically inherits from Object.prototype) and then directly accessing the '$' prefixed properties of the object. In what follows we discuss the rewriting rule for this; later we discuss how third-party code is restricted from accessing '$'-prefixed properties of any object.

We store a reference to the global object in the "$g" property of Object.prototype. Next we store a function in the "$ng" property of Object.prototype that behaves as the identity function if the argument is not a reference to the global object, and returns null otherwise. These steps are carried out by executing the following code in the global scope.

Initialization Code 1 T_{ng}

```
Object.prototype.$g = this;
Object.prototype.$ng = function(x) { x== ({ }).$g? null: x)};
```

The rewriting for this is defined as follows:

Rewrite 1 *Rewrite every occurrence of this by NOGLOBALTHIS which is defined as ({}).$ng(this).*

Wrapping Built-in functions. We wrap all built-in functions that can potentially return a reference to the global object. The following trusted initialization code demonstrates the wrapping for the method "valueOf" of Object.prototype.

Initialization Code 2 $T_{valueOf}$

```
Object.prototype.$OPvalueOf = Object.prototype.valueOf;
({ }).$OPvalueOf.call = Function.prototype.call;
Object.prototype.valueOf = function() {
                             var $= ({ }).$OPvalueOf.call(this, arguments);
                             return ($==({ }).$g?null:$);
                           };
```

The main idea is to redefine the method to a new function which calls the original "valueOf" method (stored in the property $OPvaluesOf of Object.prototype) and returns the result only if it is not a reference to the global object (stored in the property "$g" of Object.prototype). Similarly we can define the appropriate initialization codes for the methods "concat", "sort" or "reverse" of Array.prototype. We denote these by T_{sort}, T_{concat} and $T_{reverse}$.

5.3.3 Preventing Dynamic Code Generation

From Fact 3 we know that only the built-in functions @eval and @Function allow dynamic generation of code. Furthermore, in the initial heap H_0, these functions are reachable only via the properties "eval" and "Function" of the global object, and

the property "constructor" of any function object. Our approach to preventing third-party code from accessing @eval and @Function, is to conservatively disallow all access to properties "eval", "Function", and "constructor". In other words, we consider these properties as *blacklisted*. As discussed earlier, third-party code must also be prevented from accessing '$'-prefixed properties of any object. Therefore we also blacklist all '$'-prefixed property names.

As noted in Section 5.2, object properties may be accessed both implicitly and explicitly. By Fact 2 we know that only properties contained in the set $\mathcal{P}_{nat}$ can be accessed implicitly, and therefore no blacklisted property can ever get accessed implicitly. From Fact 1 we know that only expressions of the form x, $e.x$ and $e_1[e_2]$ can be used for explicitly accessing user properties. In what follows we design mechanisms for restricting the behavior of these expressions in order to prevent them from accessing blacklisted properties.

Restricting x, $e.x$. As discussed in Section 5.2, the expressions x and $e.x$ involve accessing property name "x". Therefore, in order to enforce isolation, we conservatively disallow these expressions whenever the property name "x" is blacklisted. This restriction is modeled by the following filter.

Filter 2 *Disallow all terms which contain an identifier x or expression $e.x$ such that "x" $\in$ {"eval", "Function", "constructor"} or "x" begins with the symbol '$'.*

Restricting $e_1[e_2]$. The evaluation of the expression $e_1[e_2]$ involves accessing the string form of the value obtained by evaluating expression e_2. We prevent access to blacklisted property names by rewriting every occurrence of the expression $e_1[e_2]$ to a corresponding safe expression $e_1[\mathsf{IDX}[e_2]]$ that includes a run-time check around the expression e_2 to ensure that it does not evaluate to a blacklisted property name. Every access to a blacklisted property of an object is transformed into an access to the innocuous property "bad" of the same object Although this transformation seems easy, it is complicated by subtle details of the semantics of the expression $e_1[e_2]$. The detailed steps are as follows.

We first add a method "$idx" to Object.prototype, which when takes an argument x

and returns a function-closure that when invoked returns returns the string form of x only if it is not blacklisted.

Initialization Code 3 T_{idx}

```
Object.prototype.$String = String;
Object.prototype.$idx = function(x){
                          return function(){
                                   return (x=({ }).$String(x),CHECK[x]);
                                 };
                        };
CHECK[x] := (x == "constructor" ? "bad":
              (x == "eval" ? "bad":
                (x == "Function" ? "bad":
                  (x[0] == "$" ? "bad": x)))))
```

The returned function when invoked picks the argument x from its lexical scope, converts it to a string by invoking the @String constructor (stored in the property "$String" of Object.prototype), and then returns this string is not blacklisted. The rewriting for $e_1[e_2]$ expressions is formally defined using the following rule.

Rewrite 2 *Rewrite every occurrence of* $e_1[e_2]$ *in a term by* $e_1[IDX[e_2]]$*, where,* $IDX(e_2)$ *is defined as* $\{toString: \{\}.\$idx(e_2)\}$

The above rewriting replaces the expression e_2 with an object, whose "toString" property holds the function obtained by invoking the "$idx" method of Object.prototype on the argument e_2.

The main reason why the above rewriting is complex is because it is semantics preserving for all cases when expression e_2 does not evaluate to a blacklisted string. Furthermore, we note that the simple definition $IDX[e_2]$:=$CHECK[(\{\}).\$String(e_2)$, although secure, would not be semantics preserving. This is because it would cause the expression e_2 to be converted to a string eagerly. According to the ES3 specification, the evaluation of the expression $e_1[e_2]$ involves first evaluating e_1 to a value va_1, then

evaluating e_2 to a value va_2, then converting va_1 to an object l_1 and finally converting va_2 to a string. Our rewriting preserves this evaluation order.

The name IDX in the above rewriting rule derives from a similar run-time check idx present in Facebook FBJS. Initially we intended to use the FBJS idx check in our sandboxing mechanism. However on analyzing it we found significant security vulnerabilities in it. We discuss these vulnerabilities in Section 5.4.1.

5.3.4 Formal Analysis

We now combine the filtering, rewriting and heap initialization steps to define a sandboxing mechanism for enforcing hosting page isolation. By design, these steps are all compatible with each other and can be combined in a straightforward manner. We formally prove correctness of our mechanism with respect to our operation semantics of ES3 (see Chapter 4). Given a global variable whitelist $\mathcal{G}$, we begin by defining a sub-language $ES3_{safe}(\mathcal{G})$ using the filtering steps.

Definition 3 ($ES3_{safe}$) *The sub-language $ES3_{safe}$ is defined as $Terms^{ES3}_{user}$ MINUS: all terms containing an identifier x such that "x" $\notin \mathcal{G}$ (Filter 1) or "x" begins with '$' (Filter 2), all terms containing an expression e.x such that "x" begins with '$' or "x" $\in$ { "eval", "Function", "constructor"} (Filter 2).*

Based on the rewriting steps, we define a rewriting map $\phi^h : Terms^{ES3}_{user} \rightarrow Terms^{ES3}_{user}$.

Definition 4 (ϕ^h) *Given a term $t \in Terms^{ES3}_{user}$, if $t \in ES3_{safe}$ then $\phi^h(t)$ is defined as the term obtained by recursively rewriting: (1) every occurrence of e_1[e_2] with e_1[IDX[e_2]] (Rewrite 2), and (2) every occurrence of this with NOGLOBALTHIS (Rewrite 1). For all terms $t \in ES3 \setminus ES3_{safe}$, $\phi^h(t)$ is defined as 0.*

Finally, combining all the heap initialization steps we define the initial heap H^h and stack L^h as follows. Our definition makes use of the initial ES3 heap H^{ES3}_0 and stack L^{ES3}_0 (see Chapter 4, Section 4.4.1).

Definition 5 (H^h, L^h) *The heap H^h and stack L^h are defined as the final heap and stack obtained after executing all the heap initialization codes in the global scope.*

Formally $H^h = heap(final(H_0^{ES3}, L_0^{ES3}, T_{in})$ *and* $L^h = stack(final(H_0^{ES3}, L_0^{ES3}, T_{in})$ *where* $T_{in} := T_{ng}; T_{valueOf}; T_{sort}; T_{concat}; T_{reverse}; T_{idx}$.

Note that it is very important that we execute the initialization code T_{in} on the initial heap H_0^{ES3} and stack L_0^{ES3}, and hence *before* any third-party code is executed. We now define our sandboxing mechanism $\mathcal{S}^h$ for enforcing hosting page isolation.

Definition 6 (Sandboxing Mechanism $\mathcal{S}^h$) *Given a global variable whitelist* $\mathcal{G}$*, the sandboxing mechanism* $\mathcal{S}^h$ *is defined as* $\langle H^h, L^h, \phi^h \rangle$.

We prove that $\mathcal{S}^h$ is well formed according to Definition 1.

Theorem 2 (Well-formedness of $\mathcal{S}^h$) $\mathcal{S}^h$ *is a well-formed single-component sandboxing mechanism.*

Next we formally prove that the mechanism $\mathcal{S}^h$ correctly enforces hosting page isolation for all whitelists that include the set $\mathcal{P}_{nat}$ (Assumption 1).

Theorem 3 (Correctness of $\mathcal{S}^h$) *For all global variable whitelists* $\mathcal{G}$ *such that* $\mathcal{P}_{nat} \subseteq \mathcal{G}$*, for all third-party terms* $t \in Terms_{user}^{ES3}$*, the term* $\phi^h(t)$ *satisfies the isolation property* $Isolation_{\mathcal{G}}^h(H^h, L^h, \phi^h(t))$.

The proof of this theorem is carried out by defining an invariant on program states that is preserved under reduction of all sandboxed code, and implies the hosting page isolation property. Due to the sheer volume of the semantics, we only describe a sketch of the proof in the appendix (Section A.2).

5.3.5 Discussion

We now make a few remarks about the design of our sandboxing mechanism $\mathcal{S}^h$. Isolating global variables using $\mathcal{S}^h$ involves two significant restrictions — (1) Disallowing all non-whitelisted identifier names, and (2) Disallowing all forms of dynamic code generation. The first restriction seems to make the programming model inconvenient as it forbids the use of identifiers with non-whitelisted names even if they resolve to local variables. However we argue that this inconvenience is superficial and it is

always possible to refactor the code so that it satisfies the restriction. The main idea is to have a fresh object for every lexical scope and replace all local variables with properties of the same name on this object. As a result, local variables can be accessed as object-properties which are not forced to be within the whitelist[2]. For example consider the whitelist $\mathcal{G} = \{\text{"x"}\}$. The following illegal code which makes use of non-whitelisted names "y" and "z"

```
var y = 42; z = y + 1;
```

can be refactored as

```
var x = { }; x.y = 42; x.z = x.y + 1;
```

Above, we store an empty object in the whitelisted global variable x and then replace all subsequent local variables with properties of this object.

While the restriction on dynamic code generation indeed hampers the expressivity of the language, a study by Livshits and Guarnieri [25] shows that it does not affect most third-party widgets. The study analyzes 8379 real-world widgets used on the Microsoft Vista Sidebar, Microsoft Windows Live and iGoogle and shows that only 0.4 percent of Live gadgets, and 4.7 percent of Google gadgets use the @eval function. Furthermore, for applications whose functionality depends on dynamic code generation, a restricted form of @eval can be allowed using the wrapping approach. Instead of blocking the functions @eval and @Function, we appropriately wrap them so that all input strings are first parsed, filtered and rewritten using the $\mathcal{S}^h$ sandboxing mechanism and only then converted to code. We leave the formal proof of correctness for this alternative approach to future work.

We defined the initial execution environment for $\mathcal{S}^h$ by executing the initialization code on the initial ES3 heap H_0. This makes all our formal analysis specific to the initial heap H_0, which does not include DOM. objects and additional APIs that the hosting page might have defined. However we argue that the techniques described in this chapter can be extended to other initial heaps and stacks as well. This would involve analyzing the additional objects present on these heaps, for dynamic code

[2]While object-properties are restricted by a blacklist (see Filter 2), however this restriction is relatively less significant.

generation or global object leakage channels. Once these channels have been determined they can be appropriately tamed by using similar property name blacklisting and wrapping techniques. For instance, the "getParent" method of the window object, which leaks a reference to the global object, can be wrapped using the same wrapper as that used for Object.prototype.valueOf which ensures that the return value is never a reference to the global object.

5.4 Case Study: Hosting page Isolation in FBJS

FBJS [82] is a mechanism for sandboxing third-party applications on Facebook profile pages. As discussed in Chapter 3 , FBJS is based on the API+LBS architecture and one of its main goals is to enforce hosting page isolation in order to prevent third-party applications from accessing critical resources on the main Facebook page. In the past few years, FBJS has had a huge amount of success with a large developer community writing applications compatible with FBJS that get served to millions of Facebook users. Therefore, as part of this work we analyzed the security of FBJS in detail, and compared its permissiveness of our sandboxing mechanism $\mathcal{S}^h$. We found our sandboxing mechanism to be comparable, from an application developer viewpoint, to FBJS with fewer semantic anomalies (as described below) and with the advantage of being provably safe. Our analysis also revealed multiple security vulnerabilities (reported first in [50], [48], [47]) in the FBJS enforcement of hosting page isolation. In the rest of this section we present a brief description of some of these vulnerabilities , and a detailed comparison between our mechanism and FBJS. The versions of FBJS analyzed in this work are the ones that were current in November 2008 and March 2009, denoted respectively by FBJS_{Nov08} and FBJS_{Mar09}. We refer the reader to Chapter 3 for an overview of the FBJS sandboxing mechanism.

5.4.1 Vulnerabilities Found

FBJS makes use of run-time monitoring functions ref and idx, and rewrites every occurrence of this in third-party code to ref(this) to prevent code from obtaining a

direct reference to the global object, and every occurrence of $e_1[e_2]$ to $e_1[\mathsf{idx}(e_2)]$ to prevent code from accessing blacklisted property names.

Disabling ref and idx in FBJS$_{Nov08}$. The function ref and idx are defined as methods of the global object in FBJS$_{Nov08}$. We found that it is possible for third-party code to disable these functions by shadowing them in the current scope. This allows it to obtain a direct reference to the global object and therefore access to arbitrary global variables. This vulnerability was first reported in [50]. The nature of this vulnerability can be understood by assuming that FBJS$_{Nov08}$ programs can contain an expression get_scope() which returns the current scope object; two ways of achieving this are explained later. Once a program has a handle to its own scope object, it could be disable the ref and idx functions by defining them in the current scope, such as by running get_scope().ref = function(x){return x;}.

One way to define get_scope() is by the following code

```
try {
  throw ( function(){return this;} );
} catch (get_scope) {
  var scp = get_scope(); % scp now holds a reference to the current scope object }.
}
```

In FBJS$_{Nov08}$, this code is rewritten to

```
try {
  throw ( function(){return ref(this);} );
} catch (a12345_get_scope) {
  var scp = a12345_get_scope();
  % a12345_scp now holds a reference to the current scope object.
}
```

When the code is executed, the function thrown as an exception in the try block is bound to the identifier a12345_get_scope in a new scope object that becomes the scope for the catch block. When this function gets invoked from the catch block (from the call a12345_get_scope()), the this identifier of the function gets bound to the enclosing scope object. The ref function does not filter out the reference as it is different from

the global object.

There is another, even more subtle way to access the scope object, using named recursive functions

```
var get_window = function get_scope(x) {
                   if (x==0) return this;
                   return get_scope(0); % returns a reference to the current scope object.
                 };
get_window(1);
```

Here we save a named function in a global variable named "get_window". As this function executes, the static scope of the recursive function is a fresh scope object, say o, where the identifier "get_scope" is bound to the function itself, making recursion possible. On invoking get_window(1), the else branch gets triggered which recursively calls get_scope(0). This latter function gets its this bound to the scope object o mentioned above. Since o is different from the global object, it escapes the ref check, and is returned by the recursive call get_scope(0).

The complete exploit codes for both scope-stealing mechanisms are reported in Figure 5.1. Both exploits essentially obtain a reference to the global object and then open an unauthorized pop-up dialog by invoking the "alert" method (screen shots are in Figure 5.2).

The exploit code is browser-dependent because of deviations from the ES3 specification. Safari 3.2.1 follows the specification in handling both the try-catch construct and recursive functions, and is therefore vulnerable to both attacks. Opera 9.5 and Chrome 1.0 follow the try-catch specification but depart from it on the recursive function by binding the window object instead of the scope object to this. Hence they are vulnerable to attack B only. Firefox 3.0.6 does the opposite, binding window to this in the try-catch case, and following the specification in the recursive function case. Hence, it is vulnerable to attack A only. Internet Explorer 7.0, as tested, departs from the specification binding window to this in both cases, and is therefore not vulnerable to these specific attacks.

Within hours of our disclosure to Facebook, the Facebook team addressed the

problems discussed above. To fix the problem, the team adopted our solution of separating the namespace of the run-time checks ref and idx from the namespace available to FBJS applications. This is done by adding the two functions as properties of a private object stored in the global variable "$FBJS", and preventing user code from accessing the global variable "$FBJS". FBJS$_{Mar09}$ supports the namespace-separated versions of ref and idx.

Circumventing idx in FBJS$_{Mar09}$. We now present a vulnerability that we found in the FBJS$_{Mar09}$ implementation of the $FBJS.idx runtime check. This vulnerability was first reported in [47]. The $FBJS.idx function implementation is semantically equivalent to the following code:

```
($=e2,($ instanceof Object || $blacklist[$])? "bad": $)
```

where $blacklist is the object {"eval" : true, "constructor" : true, "Function" : true}. The main problem is that the expression $blacklist[$]? "bad": $ converts $ to a string two times. This is a problem if the evaluation has a side effect. For example, the object {toString:function(){this.toString=function(){return "caller";}; return "good";}} can fool FBJS by first returning an innocuous property name "good", and then returning the blacklisted property name "caller" on the second evaluation. To avoid this problem, FBJS$_{Mar09}$ inserts the check $ instanceof Object that tries to detect if $ contains an object. Unfortunately, this check is not sound in general. According to the ES3 semantics, any object whose prototype is null (such as Object.prototype) escapes this check. We found that in Safari scope objects have their prototype set to null, and therefore we could mount attacks on $FBJS.idx that effectively let user application code escape the Facebook sandbox. Shortly after we notified Facebook of this problem, the $FBJS.idx function was modified to include a special check that ensures that if the browser is Safari then this is never bound to an object that can escape the aforesaid instanceof check. This solution is not completely satisfactory as it does not address the root of the problem. Some browsers may have host objects that have a null prototype and can be accessed without using this, thereby providing another mechanism for subverting $FBJS.idx. Our IDX expression does not suffer from such brittleness.

```
<a href="#" onclick="LM()">Test "LiveMessage" (All browsers)</a>
<script>
  var get_win = (new LiveMessage('foo')).setSendSuccessHandler;
  function LM() { get_win().alert("Hacked!");}
</script>

<a href="#" onclick="hE()"> Test "htmlEncode" (All browsers)</a>
<script>
  var get_win="foo".htmlEncode;
  function hE() { get_win().alert("Hacked!");}
</script>

<a href="#" onclick="a()">Test A (Firefox and Safari)</a>
<script>
  var get_window = function get_scope(x) {
    if (x==0) return this;
    return get_scope(0);
  };
  function a() {get_window(1).alert("Hacked!");}
</script>

<a href="#" onclick="b()"> Test B (Safari, Opera and Chrome)</a>
<script>
  function b(){
    try {
      throw (function() {return this;});
    } catch (get_scope) { get_scope().ref=function(x){return x;};
        this.alert("Hacked!")
    }
  }
</script>
```

Figure 5.1: FBJS_{Nov08} exploit codes

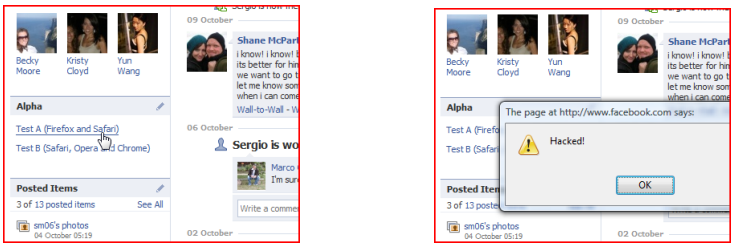

Figure 5.2: Demonstrating the FBJS_{Nov08} vulnerabilities in Firefox

5.4.2 Comparison with our mechanism

Overall FBJS imposes the same filtering and rewriting on third-party code as our sandboxing mechanism $\mathcal{S}^h$. However, there are some differences when it comes to the specific restrictions imposed on identifiers and the property access mechanism $e_1[e_2]$. Instead of disallowing identifiers corresponding to non-whielisted global variables, FBJS renames all identifiers with an application specific prefix. Besides being more permissive, this approach also serves the purpose of separating the namespaces of two different applications. However such renaming when applied to built-in properties of the global object and prototype objects, can drastically alter the semantics of code. The most obvious example is the expression toString(), that evaluates to "[object Window]" in the un-renamed version, whereas it raises a reference error exception when it is evaluated as a12345_toString() in the renamed version. The main issue is that identifier names can also resolve to properties of prototypes of scope objects. Since the built-in properties of prototype objects are not renamed, the corresponding variable names in the program should also not be renamed, in order to preserve this correspondence between them. In [50], we formalize a sufficient condition on identifier renaming functions that ensures that the semantics of the program is preserved. In [47], we develop a sandboxing mechanism that isolates global variables using identifier renaming.

As discussed in the previous section, the $FBJS.idx function in FBJS_{Mar09} is not robust and can be compromised in certain browser environments. Furthermore, the side-effect order is not preserved by the rewriting even when e_2 evaluates to a non-blacklisted property name, whereas it is preserved by our IDX rewriting. Another

minor point of difference is that our sandboxing mechanism allows third-party code to use the with construct whereas FBJS prohibits it. Our sandboxing mechanism however removes or restricts various constructs that appear in with use-cases.

In summary, our sandboxing mechanism $\mathcal{S}^h$ is very close to FBJS in the restriction it imposes on third-party code. We consider our design methodology to be a success as we were able to contribute to the security of Facebook through insights obtained by our semantic methods, and that in the end we were able to provide provable guarantees for a sandboxing mechanism of ES3 that is essentially similar to one used by external application developers for a hugely popular current site.

5.5 Related Work

In this section, we describe a few related approaches to JavaScript isolation. Some of these approaches have not been subjected to rigorous semantic analysis, and could therefore benefit from the reasoning techniques presented in this dissertation. We do not discuss FBJS and ADsafe here as those are surveyed in detail in Chapter 3. We leave the discussion on Google Caja to Chapter 6.

BrowserShield. Browsershield [70] is a system that rewrites web pages in order to enforce run-time monitoring of the embedded scripts. The systems takes an HTML page, adds a script tag to load a trusted library, rewrites embedded scripts so that they invoke a local rewriting function before being executed, and rewrites instructions to load remote scripts by making them load through a rewriting proxy. The run time monitoring is enforced by *policies* which are in effect functions that monitor the JavaScript execution. Common operations such as assignment suffer from a hundred-fold slowdown, and policies are arbitrary JavaScript functions for which there is no systematic way of guaranteeing correctness.

GateKeeper. Gatekeeper [25] propose an approach to enforcing security and reliability policies in JavaScript based on static analysis of two ES3 subsets. The first, $\mathrm{JS}_{\mathrm{Safe}}$, is obtained exclusively by filtering out with, eval, $e_1[e_2]$ or other dangerous constructs. The second subset, $\mathrm{JS}_{\mathrm{GK}}$ reinstates $e_1[e_2]$ after wrapping it in a run-time

monitor. The static analysis essentially involves extracting Datalog facts and clauses that approximate the call-graph and points-to relation of JavaScript objects at run-time[3]. The analysis necessarily loses precision in several points, and in particular when dealing with prototypes. Unfortunately, the implementation of GateKeeper is not available for inspection, and the sparse details on the definition of JS_{Safe} and the run-time monitors in JS_{GK} are not sufficient for a formal comparison with our results.

Lightweight Self-Protecting Javascript. [66] introduces a principled approach for enforcing safety properties on JavaScript built-in and host libraries (such as the DOM). The enforcement mechanism involves wrapping each of the security critical native library methods and properties before executing untrusted code. The untrusted code is not restricted at all, and all security is based on the wrappers providing a restricted execution environment for untrusted code. While this is promising approach with no pre-processing overhead for untrusted code, it is not sound for existing browsers. For example, by deleting certain properties (such as: "window") of the global object, some host objects are reinstated in the global environment, subverting the wrapping mechanism. Future versions of JavaScript may provide better support for this implementation technique.

[3]We explore a similar static analysis approach for ensuring API Confinement in Chapter 8.

Chapter 6

Mashup Isolation

In this chapter[1], we design and analyze a sandboxing mechanism for *basic mashups* in order to enforce the *Mashup Isolation* property. As introduced informally in Chapter 3, basic mashups are mashups obtained by sequentially composing third-party components obtained from multiple sources. Examples of a basic mashup include any page with multiple independent advertisements or independent social networking applications. The isolation property for basic mashups can be broken down as: (1) (*Hosting page isolation*) each mashup component only accesses certain whitelisted global variables, and (2) (*Inter-component isolation*) execution of one mashup component does not involve writing to a memory location that another component reads from.

While hosting page isolation for basic mashups can be achieved using the sandboxing mechanism designed in Chapter 5, the inter-component isolation goal still remains open. It is tempting to enforce inter-component isolation by defining disjoint global variable whitelists for third-party components and then using the hosting page isolation mechanism for enforcing these whitelists. The approach relies on the assumption that separating the global variable namespaces of two applications eliminates all communication channels between then. Unfortunately, this assumption turns out to be false as third-party components can communicate by accessing built-in objects and

[1]This chapter is based on joint work with Sergio Maffeis.

by invoking functions that side-effect built-in objects. This is illustrated by the inter-component isolation vulnerabilities that we found in FBJS during the course of this research. In this chapter, we therefore develop systematic, provably sound methods for enforcing isolation *between* two third-party components. In doing so we borrow ideas from the literature on object-capability.

Capability-based Isolation. Capability-based protection is a well known method for operating-system-level protection, deployed in such systems as the Cambridge CAP Computer, the Hydra System, StarOS, IBM System/38, the Intel iAPX423, the Amoeba operating system, and others (see [42, 80]). The main idea is that code possessing a capability, such as an unforgeable reference to a file or system object, is allowed to access the resource by virtue of possessing the capability. If a system is *capability safe,* and a process possesses only the capabilities that it is explicitly given, then isolation between two untrusted processes may be achieved by granting them capabilities with non-overlapping privilege sets.

An attractive adaptation to programming language contexts is the *object-capability model* [60, 58], which replaces the traditional subject-object dichotomy with programming language objects that are both subjects that initiate access and objects (targets) of regulated actions. Some languages that have been previously designed as object-capability languages are E [71], Joe-E [53], Emily [78], and W7 [69]. Each of these is a restriction or specialized use of a larger programming language, intended to provide capability safety by eliminating language constructs that could leak privileges beyond those entailed by the capabilities possessed by an object. Specifically, E and Joe-E are restrictions of Java, Emily is a restricted form of OCaml, and W7 is based on Scheme.

Our original goal was to systematically design an object-capability model for ES3 and then develop an inter-component isolation technique using capabilities. While developing suitable foundations for characterizing reachability and isolation in object-capability languages, we identified a concept called *authority safety*, and found that *authority isolation* is sufficient for inter-component isolation. We therefore decided to focus on developing a general theory of authority safety, and then using it to define a mechanism of enforcing authority-isolation across ES3 components.

Informally, *authority* of a term is an over-approximation of the set of all possible heap actions performed during the execution of a term, and an authority map is a mapping from terms to their respective authorities. In object-capability languages, the authority of a term is derived entirely from the capabilities possessed by it. Two access principles articulated in the object-capability literature (e.g., [58, 76, 77]) are "only connectivity begets connectivity" and "no authority amplification." Intuitively, the first condition means that all access must derive from previous access, or, if two sections of code have disjoint or "disconnected" authority then they cannot interfere with each other. The second property restricts the change in authority that may occur when a section of code executes and potentially transfers authority to another. The change in authority is limited to initial authority, authority received through interaction, and new authority obtained by allocating new resources. Since these two principles are sufficient to bound the authority of executing code, we formalize these two principles using the operational semantics and say that a mapping of authority to program terms is *safe* if these two properties are guaranteed. We give a general proof that for all basic mashups, authority isolation under a safe authority map implies inter-component isolation.

As an application of this general theory, we define a safe authority map for ES3 and develop an enforcement mechanism that achieves authority isolation between various mashup components. Our enforcement mechanism relies on a restriction on the semantics of ES3, which is that all built-in objects except the global object are *frozen*. This means that all properties of these objects are immutable, and no properties can be added or deleted from these objects.

Organization. The remainder of this chapter is structured as follows: Section 6.1 formally states the Mashup Isolation problem, and the challenges involved in solving the problem. Section 6.2 defines authority safety and sketches out a general approach for solving the mashup isolation problem using safe authority maps. Section 6.3 uses this approach for formally defining a sandboxing mechanism for enforcing mashup isolation in ES3. This is followed by a proof of correctness of the mechanism. Section 6.4 presents an authority analysis of FBJS and ADsafe, and Section 6.5 discusses related work.

6.1 The Mashup Isolation Problem

In this section we formally define the isolation problem for basic mashups written in ES3. In doing so we make use of the analysis framework for ES3 programs developed in Section 4.4. We begin with the definition of basic mashups.

6.1.1 Basic Mashups

A mashup is a composition of components, which are essentially programs labeled with unique principal ids denoting their respective sources. A basic mashup is defined as a sequential composition of mashup components. In ES3, we formalize mashup components as pairs (t, id) where $t \in Terms^{ES3}_{user}$ is a well-formed user-level ES3 term and $id \in \mathbb{I}$ is a principal id denoting the source of the components. Basic mashups are formally defined as follows:

Definition 7 (Basic Mashup) *An n-component basic mashup*

$$m := Mashup((t_1, id_1), \ldots, (t_n, id_n))$$

is an ordered list of mashup components $(t_1, id_1), \ldots, (t_n, id_n)$*. The program executed by the mashup, denoted by* $prog(m)$*, is the block statement* $\{t_1 \ldots t_n\}$*.*

Given a basic mashup $m = Mashup((t_1, id_1), \ldots, (t_n, id_n))$ we formally define the set of heap actions performed during the execution of each mashup component. In order to do this we first make the following observation.

From the semantics of ES3 [45], the evaluation trace of a block statement $\{t_1 \ldots t_n\}$ on a heap H and stack L, such that $Wf(H, L, \{t_1 \ldots t_n\})$, has the following general structure.

$$\begin{array}{rcl} H, L, \{t_1 \ldots t_n\} & \rightarrow & H, L, \mathsf{sC}[t_1] \rightarrow^* H_1, L_1, \mathsf{sC}[co_1] \\ & \rightarrow & H_1, L_1, \mathsf{sC}[t_2] \rightarrow^* H_2, L_2, \mathsf{sC}[co_2] \\ & \rightarrow & \ldots \\ & \rightarrow & H_{k-1}, L_{k-1}, \mathsf{sC}[t_k] \rightarrow^* H_k, L_k, \mathsf{sC}[co_k] \\ & \rightarrow & \ldots \end{array}$$

Above, the context sC is a statement context for evaluating statements. Thus during the execution of the statement $\{t_1 \ldots t_n\}$, terms $t_1, \ldots, t_n$ execute in sequence and the

execution of a term t_i starts on the final heap H_{i-1} and stack L_{i-1} obtained after executing terms $t_1, \ldots, t_{i-1}$. It is possible that the execution of a term t_k diverges or generates an error. In such a case, terms $t_{k+1}, \ldots, t_n$ do not get a chance to execute. In general for a block statement $\{t_1 \ldots t_n\}$, heap H and stack L, we define $Abnormal(H, L, \{t_1 \ldots t_n\})$ as the smallest natural number k such that the execution of t_k diverges or generates an error. If the executions of all the terms terminate normally then $Abnormal(H, L, \{t_1 \ldots t_n\})$ is $n + 1$. Furthermore, for all i such that $2 \leq i \leq Abnormal(H, L, \{t_1 \ldots t_n\})$, we define $HS(H, L, \{t_1 \ldots t_n\}, i)$ as the initial heap and stack when t_i begins execution (same as the final heap H_{i-1} and stack L_{i-1} obtained after executing terms $t_1, \ldots, t_{i-1}$). $HS(H, L, \{t_1 \ldots t_n\}, 1)$ is defined as H, L.

We are now ready to define the map $MAct(H, L, m, i)$ which denotes the set of heap actions performed during the execution of the i^{th} component when the mashup program $prog(m)$ begins execution at the heap H and stack L.

Definition 8 ($MAct(H, L, m, i)$) *Given an n-component basic mashup $m := Mashup((t_1, id_1), \ldots, (t_n, id_n))$, a heap H and stack L, such that the state $H, L, prog(m)$ is well formed, $MAct(H, L, m, i)$ is defined as $Act(\tau(H_{i-1}, L_{i-1}, t_i))$ if $i \leq Abnormal(H, L, prog(m))$ and the empty set $\emptyset$ for $i > Abnormal(H, L, prog(m))$, where $H_{i-1}, L_{i-1} = HS(H, L, \{t_1 \ldots t_n\}, i)$.*

6.1.2 The "Can-Influence" Relation

We now define a relation $\triangleright$, read as "can influence", on heap actions. Recall from Chapter 4, Section 4.4 that a heap action is a triple l, p, d consisting of a location l, property name p and an access-descriptor d indicating whether property p of location l is read or written. The purpose of the "can-influence" relation is to determine whether one heap action can directly influence another heap action. Concretely, we say that an action a_1 "can-influence" an action a_2, if a_1 involves writing to a particular location-property, and a_2 involves reading from the same location-property.

Definition 9 (Can Influence) *An action a_1* can influence *action a_2, written as $a_1 \triangleright a_2$, iff $loc(a_1) = loc(a_2)$, $props(a_1) = props(a_2)$, $perm(a_1) = \mathsf{w}$ and $perm(a_2) = \mathsf{r}$.*

A set A_1 of heap actions can influence *set A_2, written $A_1 \triangleright A_2$ if $\exists a_1 \in A_1, a_2 \in A_2$ such that $a_1 \triangleright a_2$.*

6.1.3 Sandboxing Mechanism

We now formally define language-based sandboxing mechanisms for basic mashups. The mechanism is based on a combination of filtering and rewriting of third-party code and wrapping of security-critical hosting page resources.

Definition 10 (Mashup Sandboxing Mechanism) *A language-based sandboxing mechanism for a n-component basic mashup is formalized as a tuple $\langle H, L, \phi_1, \ldots, \phi_n \rangle$ consisting of a heap H, stack L and functions $\phi_1, \ldots, \phi_n$ from $Terms^{ES3}_{user}$ to $Terms^{ES3}_{user}$. The mechanism is well formed iff the heap H and stack L are well-formed, and for all terms $t \in Terms^{ES3}_{user}$, if t is well-formed with respect to the heap H then for each $i \in \{1, \ldots, n\}$, $\phi_i(t)$ is also well-formed with respect to the heap H.*

Similar to single-component sandboxing mechanisms discussed in Chapter 5, the heap H and stack L model the initial execution environment for the mashup with security-critical resources wrapped, and the maps $\phi_1, \ldots, \phi_n$ model the filtering and rewriting applied to third-party components. Given a well-formed sandboxing mechanism $\mathcal{S}^m := \langle H, L, \phi_1, \ldots, \phi_n \rangle$ and a mashup $m = \mathit{Mashup}((t_1, id_1), \ldots, (t_n, id_n))$, the "sandboxed" mashup is defined as $\mathcal{S}^m\langle m \rangle := \mathit{Mashup}((\phi_1(t_1), id_1), \ldots, (\phi_n(t_n), id_n))$. The initial execution state of the sandboxed mashup is $H, L, \mathit{prog}(\mathcal{S}^m\langle m \rangle)$.

6.1.4 Problem Statement

In order to formally define the Mashup Isolation problem, we first define an isolation property for basic mashups. Informally, the property consists of two parts: (i) the actions performed by the individual components are mutually non-influencing; (ii) the set of actions performed by each component do not include accessing global variables outside a given whitelist. In our definition, we use $\mathcal{G}$ to denote a whitelist of global variables. We abuse notation and use the same name $\mathit{Isolation}^m_{\mathcal{G}}$ from Chapter 5 for the isolation property for mashups. Since this definition has a different type signature from the one in Chapter 5 the ambiguity should be resolvable from the context.

Definition 11 (Mashup Isolation Property) *Given a whitelist $\mathcal{G}$ of global variables, a basic mashup $m = Mashup((t_1, id_1), \ldots, (t_n, id_n))$ achieves mashup isolation for a heap H and stack L iff $Wf(H, L, prog(m))$ implies the following properties:*

1. $\forall i, j : i < j \Rightarrow MAct(H, L, m, i) \not\triangleright MAct(H, L, m, j)$
2. $\forall i : \forall a : (a \in MAct(H, L, m, i) \ \wedge \ loc(a) = \#global) \implies props(a) \in \mathcal{G}$

The property is formally denoted by the predicate $Isolation^m_{\mathcal{G}}(H, L, m, \mathcal{G})$.

We now formally state the mashup isolation problem.

> *Given a whitelist $\mathcal{G}$ of global variables, design a well-formed sandboxing mechanism $\mathcal{S}^m := \langle H, L, \phi_1, \ldots, \phi_n \rangle$ such that for all n-component mashups m, the mashup $\mathcal{S}^m\langle m \rangle$ achieves mashup isolation.*

6.1.5 Challenges and Approach

The central challenge in solving the mashup isolation lies in designing a sandboxing mechanism $\mathcal{S}^m$ that meets the inter-component isolation goal. For a sandboxed mashup $\mathcal{S}^m\langle m \rangle$, this goal is formally stated as:

$$\forall i, j : i < j \Rightarrow MAct(H, L, \mathcal{S}^m\langle m \rangle, i) \not\triangleright MAct(H, L, \mathcal{S}^m\langle m \rangle, j)$$

Achieving this goal requires reasoning about the set $MAct(H, L, \mathcal{S}^m\langle m \rangle, i)$, that is, the set of actions performed by component i during the execution of the mashup $\mathcal{S}^m\langle m \rangle$. This is where the two main challenges lie: (i) precisely reasoning about the set of *all* actions performed by component i requires code analysis which is challenging due to the various dynamic features of ES3, and (ii) the heap and stack with respect to which component i executes comes into existence only *after* components $1, \ldots, i-1$ have finished execution, and therefore are unknown statically.

We overcome both these challenges by using the concepts of *authority of a term* and *authority safety*, obtained from the object-capability literature [60, 58]. Informally, the authority of a term for a given heap and stack is an over-approximation of the set of actions performed by the term when executed with respect to the given

heap and stack. In Section 6.2 we show that for a safe authority map if the *authorities* of the individual mashup components do not influence each other on the initial heap H and stack L then they will also not influence each other on any intermediate heap H_i and stack L_i obtained during the execution of the mashup. Thus in order to satisfy the inter-component isolation goal we only need a safe authority map that ensures that the authorities of the individual sandboxed components do not influence each other on the initial heap H and stack L (which is known statically). In the next section, we formalize the concept of authority safety.

6.2 Authority Safety

We formally define the concepts of authority safety and authority isolation, and show that authority isolation under a safe authority map is sufficient to ensure mashup isolation, as characterized in Section 6.1.4. We first explain the concepts informally for general sequential imperative programming languages and then develop the formal machinery for ES3.

In programming languages, *subjects* are program terms, and *resources* are the smallest granularity readable/writable positions on the program heap. Given an initial heap and stack, an authority map provides a mapping between the subject and an upper bound on the set of actions performed when the subject executes, starting with given heap and stack.

Authority Safety. In an object-capability system, a subject derives all its authority from the set of capabilities held by it. Such authority maps satisfy two fundamental properties:

(1) **Only connectivity begets connectivity:** A subject can influence the authority of only those subjects whose authority influences its own authority.

(2) **No authority amplification:** The change in authority of a subject due to actions performed by another subject is bounded by the authority of the acting subject.

The above properties together constitute *authority safety*. Although derived from the object-capability model, authority safety is actually independent of the specific details of the object-capability model, and can be independently evaluated for a given authority map. After defining authority safety more precisely below, we prove that for any two subjects t_1 and t_2 provided by principals id_1 and id_2, if the initial authorities of t_1 and t_2 are non-influencing, then the evaluation of term t_1 cannot influence the evaluation of term t_2 in the mashup $Mashup((t_1, id_1), (t_2, id_2))$. This result allows us to show that inter-component isolation can be achieved by sandboxing third-party components such that their initial authorities are mutually non-influencing.

6.2.1 Formalizing Authority safety for ES3

We now define authority maps and authority safety for ES3. Our definition makes use of the map $Actions(H)$ (defined in Chapter 4, Section 4.4) which for a heap H gives the set of all valid heap actions. Formally,
$Actions(H) := \{l, p, d \mid l \in dom(H) \ \wedge \ p \in Props \ \wedge d \in \mathbb{D}\}$. An authority map $Auth$ is function with the signature $Heaps^{ES3} \times Stacks^{ES3} \times Terms^{ES3}_{user} \rightarrow Actions$. For any heap H, stack L and term t, $Auth(H, L, t) \subseteq Actions(H)$.

Definition 12 (Authority Safety) *An authority map Auth is safe iff for all well-formed states H, L, t where $t \in Terms^{ES3}_{user}$, the following holds:*

1. ***Sufficiency***
$$Act(\tau(H, L, t)) \cap Actions(H) \subseteq Auth(H, L, t).$$

The following properties apply only when the reduction trace for H, L, t terminates to a final state, say, H_f, L_f, co_f.

2. ***Only Connectivity begets connectivity:*** *For all terms $u \in Terms^{ES3}_{user}$,*
$$Act(\tau(H, L, t)) \not\triangleright Auth(H, L, u) \implies Auth(H, L, u) = Auth(H_f, L_f, u).$$

3. ***No authority amplification:*** *For all terms $u \in Terms^{ES3}_{user}$,*

$$acc(\tau(H, L, t)) \triangleright Auth(H, L, u) \Longrightarrow Auth(H_f, L_f, u) \subseteq \begin{pmatrix} Auth(H, L, u) \cup Auth(H, L, t) \\ \cup \ Actions(H_f) \setminus Actions(H) \end{pmatrix}.$$

6.2.2 Solving the Mashup Isolation Problem

We show that safe authority maps can be used to solve the Mashup Isolation problem. Given a safe authority map, authority isolation holds for basic mashups if the initial authorities of mashup components do not influence another, and donot include any actions that involve accessing global variables outside a given whitelist.

Definition 13 (Authority Isolation) *Given a whitelist $\mathcal{G}$ of global variables, a basic mashup $m = Mashup((t_1, id_1), \ldots, (t_n, id_n))$ achieves authority isolation for a heap H, stack L, iff $Wf(H, L, prog(m))$, implies there exists a safe authority map $Auth$ such that the following properties hold:*

1. $\forall i, j : i < j \Rightarrow Auth(H, L, t_i) \not\triangleright Auth(H, L, t_j)$.
2. $\forall i : \forall a : (a \in Auth(H, L, t_i) \ \wedge \ loc(a) = \#global) \implies props(a) \in \mathcal{G}$.

The property is formally denoted by the predicate $AuthIsolation(H, L, m, \mathcal{G})$.

We now show that *AuthIsolation* implies *Isolation* for all basic mashups.

Theorem 4 *For all basic mashups $m = Mashup((t_1, id_1), \ldots, (t_n, id_n))$, all global variable whitelists $\mathcal{G}$, heaps H, stacks L:*

$$AuthIsolation(H, L, m, \mathcal{G}) \implies Isolation^m_{\mathcal{G}}(H, L, m, \mathcal{G}).$$

The proof of this theorem is described in detail in the appendix (Section A.3). Using this theorem, the Mashup Isolation problem for a n-component basic mashup m, can be reduced to defining a sandboxing mechanism $\mathcal{S}^m := \langle H, L, \phi_1, \ldots, \phi_n \rangle$ such that authority isolation holds for the mashup $\mathcal{S}^m\langle m \rangle$, heap H and stack L.

6.3 Sandboxing Mechanism

In this section, we design a sandboxing mechanism for solving the Mashup Isolation problem. In light of the result in Section 6.2, we design the mechanism such that for any basic mashup, authority isolation holds for the sandboxed mashup in the initial environment specified by the mechanism. We begin by giving a high-level overview of the mechanism and then dive into the details.

6.3.1 Overview

In order to achieve authority isolation, the sandboxing mechanism must ensure that the authority of each sandboxed component, in the initial execution environment: (i) does not involve accessing any non-whitelisted global variable, and (ii) does not influence the authorities of other sandboxed components. In order to achieve goal (i), we begin with the sandboxing mechanism $\mathcal{S}^h := \langle H^h, L^h, \phi^h \rangle$ defined in Chapter 5 for enforcing a global variable whitelist. We rewrite each mashup component t_i, id_i using the unction ϕ^h to $\phi^h(t_i), id_i$.

To achieve goal (ii), we first ensure that different mashup components cannot access the same global variables. This is done by prefixing all identifier names x appearing in each mashup component $\phi^h(t_i), id_i$ with the string "id_i". We assume that the resulting prefixed identifier names are also covered by the whitelist $\mathcal{G}$. This is a reasonable assumption as it is unlikely for a hosting page to have a critical object reference stored in or reachable from a id_i prefixed global variable.

Unfortunately authority isolation still does not hold for the basic mashup formed from components $\alpha_i(\phi^h(t_i)), id_i$. This is because the components can influence each other via the built-in objects. For instance, two components can communicate by writing and reading properties of the built-in Object.prototype.toString object.

Component 1: {}.toString.channel = "message";

Component 2: {}.toString.channel; *% returns "message".*

Furthermore, certain built-in methods like "push" and "pop" of Array.prototype implicitly write and read properties of the global object respectively. Such functions can also be used as a communication mechanism by mashup components. For instance,

Component 1: [ ].push("message");

Component 2: [ ].pop(); *% returns "message".*

Above, Component 1 implicitly updates the property "0" of the global object and Component 2 reads it.

In order to prevent components from sharing authority over built-in objects, we initialize the execution environment H^h, L^h for the mashup by making all objects transitively reachable from the built-in objects readonly, and wrapping all built-in methods so that they do not implicitly access properties of the global object. We

denote this restricted execution environment by $H^m_{\mathcal{G}}, L^m_{\mathcal{G}}$ and define the sandboxing mechanism $\mathcal{S}^m_{\mathcal{G}} := \langle H^m_{\mathcal{G}}, L^m_{\mathcal{G}}, \alpha_i \circ \phi^h, \ldots, \alpha_n \circ \phi^h \rangle$. We then formally show that for any n-component basic mashup m, the sandboxed mashup $\mathcal{S}^m_{\mathcal{G}}\langle m \rangle$ achieves authority isolation for the heap $H^m_{\mathcal{G}}$, stack $L^m_{\mathcal{G}}$ and the global variable whitelist $\mathcal{G}$.

6.3.2 Prefixing Identifier Names

We prefix the identifiers appearing in each mashup component using the following rewrite rule.

Rewrite 3 *Rewrite every identifier x in a mashup component from source id_i with id_i::x where :: is the string concatenation operator.*

Next we formally state our assumption on the whitelist $\mathcal{G}$.

Assumption 2 *The whitelist $\mathcal{G}$ includes all names that begin with one of the prefixes $id_1, \ldots, id_n$.*

6.3.3 Restricting Builtin Objects

Wrapping built-in functions. Mashup components influence each other by invoking built-in functions that implicitly access the global object. In order to prevent this we wrap all built-in functions so that their this argument is never the global object. This is done by running the following initialization code, similar to the one used in Section 5.3. Below, $e.x$ is a reference to a built-in function, as in Object.prototype.toString. We assume that "$e_x" is string unique to each such a reference.

Initialization Code 4 $T_{e.x}$.

```
Object.prototype.$e_x = e.x; % third-party code is disallowed from accessing $-variables
({ }).$.e_x.call = Function.prototype.call;
e.x = function(){
        var $this = (this==({ }).$g? { }: $this);
        return ({ }).$e_x.call($this);
      };
```

Let $Refs_{builtin}(H^h)$ be the set of references to all built-in function objects in the heap H^h. We denote by T_{wrap}, the sequential composition of initialization codes for all references $e.x \in Refs_{builtin}(H^h)$.

Freezing built-ins. Mashup components can also communicate with each other by reading or writing properties of commonly reachable objects. In order to prevent this, we take the approach of *freezing* all such objects, that is, we make all their properties read-only and prevent adding or deleting properties from them. Unfortunately, in ES3 it is not possible to freeze an object. However most browsers implementing ES3 support Setters and Getters (see Chapter 2), using which one can simulate object freezing. Moreover, the version of EcmaScript after ES3, namely ES5, explicitly provides a built-in function Object.freeze that can be used to freeze objects.

In order to augment ES3 with the functionality to freeze objects we propose adding a special statement freezeAll to the language that freezes all objects present on the heap except the global object. The reduction rule for freezeAll is defined as follows. We mark an object as frozen by setting a special internal property @frozen on it.

$$\frac{\begin{array}{c} dom(H_1) = dom(H) \\ \text{for each } l \in dom(H) \setminus \{\#\mathsf{global}\} : H_1(l) = H(l)[\mathsf{@frozen} \rightarrow \mathsf{true}] \\ H_1(\#global) = H(\#global) \end{array}}{H, L, \mathsf{freezeAll} \xrightarrow{s} H_1, L, (\mathsf{Normal},\ \mathsf{@empty},\ \mathsf{@empty})}$$

The reduction rule for the internal expression ł.@Put(m,va) is accordingly modified to account for frozen objects.

$$\frac{H(l).\mathsf{@frozen} = true}{H, L, \text{ł.@Put}(m,va) \xrightarrow{e} H, L, va}$$

In the rest of this Chapter we only consider the language $ES3$ augmented with the freezeAll statement. We note that the analysis framework defined for ES3 in Chapter 4 can be adapted to account for the freezeAll statement. In particular, the grammar for internal and user terms now also includes the statement freezeAll. Therefore the definitions of $Terms^{ES3}$ and term well-formedness are appropriately modified. Furthermore, the sandboxing mechanism $\mathcal{S}^h$ defined in Chapter 5 is unaffected by the

freezeAll statement. The intuition is that using the freezeAll does not grant any additional privileges to third-party code[2]. Henceforth, we will only focus on the language ES3 augmented with the freezeAll statement.

After running the initialization for wrapping built-in functions, the following initialization must be run to freeze all commonly reachable objects.

Initialization Code 5 (T_{freeze}) *freezeAll;*

6.3.4 Formal Analysis

We now formally define and analyze the sandboxing mechanism for achieving mashup isolation. The first step of our mechanism is to filter and rewrite all third-party components using the function ϕ^h obtained from the sandboxing mechanism $\mathcal{S}^h := \langle H^h, L^h, \phi^h \rangle$ defined in Chapter 5 for achieving hosting page isolation. The next step is to rename all identifiers appearing in all third-party components according to the Rewrite rule 3. In order to do this, we define a function $\alpha_i : Terms^{ES3}_{user} \rightarrow Terms^{ES3}_{user}$ as follows:

Definition 14 (α_i) *Given a term $t \in Terms^{ES3}_{user}$, $\alpha_i(t)$ is defined as the term obtained by replacing every identifier x in t with the identifier id_i::x where id_i is the source identifier associated with the i^{th} mashup component.*

Next we define the heap $H^m_{\mathcal{G}}$ and stack $L^m_{\mathcal{G}}$ using the heap initialization codes.

Definition 15 ($H^m_{\mathcal{G}}, L^m_{\mathcal{G}}$) *The $H^m_{\mathcal{G}}, L^m_{\mathcal{G}}$ are the final heap and stack obtained by running the initialization code $T_{wrap}; T_{freeze}$ on the heap H^h and stack L^h. Formally,*
$H^m_{\mathcal{G}} = heap(final(H^h, L^h, T_{wrap}; T_{freeze}))$ *and*
$L^m_{\mathcal{G}} = stack(final(H^h, L^h, T_{wrap}; T_{freeze}))$.

We now formally define the sandboxing mechanism $\mathcal{S}^m_{\mathcal{G}}$.

Definition 16 (Sandboxing Mechanism $\mathcal{S}^m_{\mathcal{G}}$) *The sandboxing mechanism $\mathcal{S}^m_{\mathcal{G}}$ is defined as $\langle H^m_{\mathcal{G}}, L^m_{\mathcal{G}}, \alpha_i \circ \phi^h, \ldots, \alpha_n \circ \phi^h \rangle$.*

[2]Rather executing freezeAll de-escalates the privileges held by code.

We prove that the mechanism $\mathcal{S}^m_{\mathcal{G}}$ is well formed according to definition 10.

Theorem 5 *$\mathcal{S}^m_{\mathcal{G}}$ is a well-formed mashup sandboxing mechanism.*

We now show that for any n-component basic mashup m, the sandboxed mashup $\mathcal{S}^m_{\mathcal{G}}\langle m\rangle$ achieves authority isolation for the initial execution environment $H^m_{\mathcal{G}}, L^m_{\mathcal{G}}$. The mashup isolation property for $\mathcal{S}^m_{\mathcal{G}}\langle m\rangle$ then follows immediately from Theorem 11. We begin by defining a safe authority map $Auth^m$ for the $ES3$ augmented with the freezeAll statement.

Authority map $Auth^m$. Since, we are only interested in using the authority map $Auth^m$ for establishing isolation for mashups sandboxed under the mechanism $\mathcal{S}^m_{\mathcal{G}} := \langle H^m_{\mathcal{G}}, L^m_{\mathcal{G}}, \alpha_i \circ \phi^h, \ldots, \alpha_n \circ \phi^h\rangle$, we only give a nontrivial definition to $Auth^m$ on states that appear during the execution of maps sandboxed using the mechanism $\mathcal{S}^m_{\mathcal{G}}$. Such states are called $\mathcal{S}^m_{\mathcal{G}}$-*consistent* and are defined as follows. We first note that $\alpha_i \circ \phi^h = \phi^h \circ \alpha_i$ and therefore all sandboxed terms belong to the set $codom(\phi^h)$. A state S is $\mathcal{S}^m_{\mathcal{G}}$-consistent iff $term(S) \in Terms^{ES3}_{user}$ and there exists a term t of the form $\phi^h(t')$ for some $t' \in Terms^{ES3}_{user}$, such that $H^m_{\mathcal{G}}, L^m_{\mathcal{G}}, t \rightarrow^* S$.

For all states H, L, t that are not $\mathcal{S}^m_{\mathcal{G}}$-*consistent*, $Auth^m(H, L, t)$ is conservatively defined as the top set $Actions(H)$. In order to define the map $Auth^m$ on $\mathcal{S}^m_{\mathcal{G}}$-*consistent* states, we first set up some definitions.

Definition 17 (Heap Graph) *Given a well-formed heap H, we denote by $Gr(H)$ a directed graph consisting of heap locations as nodes and an edge from node l_i to l_j labelled with $p \in Props$ iff $H(l_i).p = l_j$. (Note that p may also be an internal property name.)*

Given a heap graph G, a heap address l and a set of property names $P \subseteq Props$, we denote by $Reach(G, l, P)$ the set of heap addresses reachable from location l by accessing property names from P. We abuse notation and naturally extend the $Reach$ function to sets of heap locations. We also define $Next(G, l, P)$ as the set of heap locations that are connected to l by a single edge with label in the set P.

Before describing the authority map we recall a few properties about the sandboxing mechanism $\mathcal{S}^h$ form Chapter 5. $\mathcal{S}^h$ prevents access to blacklisted properties

"eval", "Function", "constructor" and all '$'-prefixed property names. We use $\mathcal{B}$ to denote the set of all blacklisted property names. $\mathcal{S}^h$ also prevents access to all global variables outside the whitelist $\mathcal{G}$. We assume that the set $\mathcal{P}_{nat}$ of implicitly accessible properties are whitelisted by default and included in the set $\mathcal{G}$.

Without accounting for the freezing of objects carried out by the initialization code freezeAll, the authority map for a $\mathcal{S}^m_{\mathcal{G}}$-*consistent* heap H, stack L, and a term $t \in codom(\phi^h)$ can be informally described as the set of all actions (l, p, d) such that one of the following holds:

(1) $d = \mathsf{r}$; $l = \#global$; p is an internal property or a property from the set $\mathcal{P}_{nat}$;

(2) $d = \mathsf{r}$ or w; $l = \#global$; p is an identifier present in the term t

(3) $d = \mathsf{r}$; $l \neq \#global$ and l is reachable, in graph Gr^H, from $\#global$ or a location in L or t, via non-blacklisted property names; p is a non-blacklisted property name.

(4) $d = \mathsf{r}$ or w; $l = \#global$; p is an identifier name appearing in a function stored at a heap location reachable, in graph Gr^H, from $\#global$ or from a location in L or t, via non-blacklisted property names.

To account for the effect of the initialization code freezeAll, we subtract the actions $(dom(H^m_{\mathcal{G}}) \setminus \{\#global\} \times Props_{user}) \times \{\mathsf{w}\}$ from the output of this authority map. The formal definition of the authority map $Auth^m$ is given below. Given a term t, $\mathcal{N}(t)$ denotes the set of identifier names appearing in a term t. For a heap H and set of locations $\mathcal{L}$, $Funcs_H(l)$ is the set of locations $l \in \mathcal{L}$ that correspond to function objects (i.e. @call $\in H(l)$)

Definition 18 ($Auth^m$) *For all states H, L, t that are not $\mathcal{S}^m_{\mathcal{G}}$-consistent, $Auth^m(H, L, t)$ is defined as $Actions(H)$. For state H, L, t that are $\mathcal{S}^m_{\mathcal{G}}$-consistent, $Auth^m(H, L, t)$ is defined as*

$$(A_1 \cup IdAuth_H(H, L, \mathcal{N}(t), \emptyset)) \setminus A_{\not\mathsf{w}}$$

where $A_{\not\mathsf{w}}$, A_1, and $IdAuth_H$ are

$$\begin{aligned} A_{\not\mathsf{w}} &:= \big(dom(H^m_{\mathcal{G}}) \setminus \{\#global\}\big) \times Props \times \{\mathsf{w}\} \\ A_1 &:= \{\#global\} \times (\mathcal{P}_{nat} \cup Props_{int}) \times \{\mathsf{r}\} \end{aligned}$$

and for sets P and P_{done} of identifier names, $IdAuth_H(H, L, P, P_{done})$ is computed using the following recursive procedure:

(1) *If $P \subseteq P_{done}$ return $\emptyset$*

(2) $Ł_1 := Next(Gr(H), L, P \cup \mathcal{P}_{nat} \cup Props_{int})$

(3) $A_i := Ł_1 \setminus \{\#global\} \times P \cup \mathcal{P}_{nat} \cup Props_{int} \times \{\mathsf{r}, \mathsf{w}\}$

(4) $A_{ii} := \{\#global\} \times P \times \{\mathsf{r}, \mathsf{w}\}$

(5) $Ł_2 := Reach(Gr(H), Ł_1, Props \setminus \mathcal{B})$

(6) $A_{iii} := (Ł_2 \setminus \{\#global\}) \times (Props \setminus \mathcal{B}) \times \{\mathsf{r}, \mathsf{w}\}$

(7) $P_1 := \{\mathcal{N}(H(l).\texttt{@body}) \mid l \in Funcs_H(Ł_2)\}$

(8) *return* $A_i \cup A_{ii} \cup A_{iii} \cup IdAuth_H(H, L, P_1, P \cup P_{done})$

Theorem 6 *$Auth^m$ is a safe authority map for the language $ES3$ augmented with the* freezeAll *statement.*

The proof of this theorem is carried out by defining an inductive state invariant that is preserved under reduction, and ensures that all actions performed during reduction are contained in the initial authority. The proof is described in detail in the appendix (Section A.3).

Authority isolation for sandboxed mashups. Using the authority map $Auth^m$ we show that for all mashups m, the sandboxed mashup $\mathcal{S}^m_{\mathcal{G}}\langle m \rangle$ achieves authority isolation for the language $ES3$ augmented with the freezeAll statement. We assume the following about the whitelist $\mathcal{G}$: (1) ($\mathcal{P}_{nat}$-compatibility) the set of implicitly accessed properties $\mathcal{P}_{nat}$ is a subset of the whitelist $\mathcal{G}$, and (2) (id-compatibility) for any i, all id_i prefixed identifier names are whitelisted in $\mathcal{G}$. These assumptions would hold true for most practical whitelists $\mathcal{G}$.

Theorem 7 *For all basic mashups m and for all $\mathcal{P}_{nat}$-compatible and id-compatible whitelists $\mathcal{G}$, the mashup $\mathcal{S}^m_{\mathcal{G}}\langle m\rangle$ achieves authority isolation for the language ES3 augmented with the freezeAll statement.*

The proof of this theorem follows from the definition of the map the $Auth^m$ and is described in detail in the appendix (Section A.3).

6.3.5 Discussion

We remark that the concept of authority safety and its application to the mashup isolation problem was first formalized in [49]. The results in [49] are very general and apply to any imperative programming language that supports sequential composition. Besides formalizing authority safety, [49] also formalizes object-capability-safety for programming languages, and proves object-capability-safety for a core subset of *Cajita* which is a sub-language of JavaScript defined by the Google Caja framework. The results developed in this chapter are essentially applications of the theory of authority safety and authority isolation to ES3.

6.4 Authority Analyses of FBJS and ADsafe

In the previous section we presented a solution to the mashup isolation problem based on isolating authorities of various mashup components. In this section we show that authority analysis is a powerful technique for analyzing security properties of sandboxing mechanisms in general. Examining the authority of code sandboxed by a mechanism may reveal access privileges that the mechanism fails to curtail. We carried out an authority analysis of the Yahoo! ADsafe and Facebook FBJS sandboxing mechanisms and identified multiple security vulnerabilities (reported first in [49]) in both of them, including an inter-component isolation failure in FBJS. The versions of ADsafe and FBJS analyzed are the ones that were current in November 2009, and are denoted by ADSafe_{Nov09} and FBJS_{Nov09} respectively.

6.4.1 FBJS

The FBJS sandboxing mechanism is used by Facebook in order to sandbox third-party applications on user profile pages. It follows the API+LBS architecture described in Chapter 3. The API is the FBJS DOM API and the sandboxing mechanism is essentially FBJS subset of JavaScript defined using the a combination of syntactic checks and source-to-source rewriting. We examined the authority available to FBJS_{Nov09} expressions and found both an inter-component isolation failure and a hosting page isolation failure in the mechanism. These vulnerabilities have been reported to Facebook, and have since then been fixed. Below, we describe these vulnerabilities in detail. We refer the reader to Chapter 3 for an overview of the FBJS sandboxing mechanism.

Inter-component isolation failure. While trying to define the authority provided to FBJS_{Nov09} expressions, we found that the actual FBJS_{Nov09} implementation does not take into account the authority conveyed by the prototype-based inheritance mechanism. Two Facebook applications can easily (and unsafely) exchange data or code with each other, bypassing the FBJS runtime altogether. The sending application can execute the code

```
({}).toString.channel = "message";
```

This code stores "message" in the property "channel" of the built-in "toString" method of Object.prototype. The receiving one can read the data back by simply reading the property.

```
var a67890_message = ({}).toString.channel;
```

Thus this example reveals an inter-component isolation failure in FBJS_{Nov09}. While this communication mechanism could be considered useful by application writers, it is not a documented FBJS_{Nov09} feature, and we do not believe it to be intended by the FBJS_{Nov09} designers. Such communication channels are not possible within our sandboxing mechanism $\mathcal{S}^m_{\mathcal{G}}$ as we explicitly forbid adding, deleting and modifying properties of all built-in prototype objects.

Hosting page isolation failure. A programming pattern common in both Face-

```
<a href="#" onclick="break()">Attack FBJS!</a>
<script>
function break() {
  var f = function() { };
  f.bind.apply = (function(old) {
                    return function(x,y){
                              var getWindow = y[1].setReplay;
                              getWindow(0).alert("Hacked!"); % access global variable
alert.
                              return old(x,y);
                            };
                  })(f.bind.apply);
}
</script>
```

Figure 6.1: FBJS_{Nov09} exploit code

book applications consists of calling functions through the standard "call" and "apply" methods. Another common idiom is to use a "bind" method to curry the arguments of a function.

We ask the question: *What if a malicious application redefines the "apply" method?* Then, the attacker can automatically hijack all the function invocations arising from expressions of the form f.bind.apply(e), and have full access to the argument e. This is indeed the basis of our attack shown in Figure 6.1. The attack code redefines the "apply" method of the "bind" function to a malicious function that steals the object passed as the second argument and accesses certain specific properties of it. Whenever an Facebook page receives an AJAX message, it invokes the "apply" method of the "bind" function, and passes it a specific object as the second argument, from which our malicious function can obtain a direct reference to the global object (window object), and as a result access all global variables. The details of the specific Facebook library function responsible for responding to AJAX messages are complicated and not necessary for the current discussion. The important point is that untrusted code is not supposed to have the ability to modify properties of built-in prototype objects.

6.4.2 ADsafe

The ADsafe [15] sandboxing mechanism is designed to protect hosting pages from embedded third-party advertising code. It also follows the API+LBS architecture described in Chapter 3. The API is the ADsafe DOM API and the sandboxing mechanism is the JSLint static analyzer. We examined the authority of adverstisement code that passes through JSLint and gets exposed to the ADsafe DOM API, in the November 2009 version of ADsafe, namely ADSafe$_{Nov09}$. Our analysis revealed three hosting page isolation failures that allow untrusted adverstisement code to read arbitrary global variables in the hosting page's execution environment. We disclosed these vulnerabilities to the primary author of the ADsafe API — Douglas Crockford, who promptly fixed them. The vulnerabilities are described below. The object dom in all code below represents the ADSafe$_{Nov09}$ DOM API. We refer the reader to Chapter 3 for an overview of the ADsafe sandboxing mechanism.

Hosting page isolation failures. The first problem we found is that the library function dom.tag exposed to untrusted code returns an object with a method "ephemeral" that returns its this parameter. This method therefore if invoked in the global scope returns a reference to the global object from which arbitrary global variables are accessible. The following code passes throught JSLint, and obtains a reference to the global object and accesses the global variable alert.

```
var a = dom.tag("div").ephemeral;
var asd = a().alert("Hacked!");
```

Any safe authority analysis of *JSLint*-passing code would immediately reveal the authority to read the ADSafe$_{Nov09}$ library method dom.tag, and, transitively, the authority to read a reference to the global object.

The second problem we found is related to the attack on the FBJS$_{Mar09}$ idx function reported in Chapter 5, and exploits the implicit type conversion mechanism. The dom.tag library function, which is available to advertisement code for creating any non-dangerous DOM elements, can be forced to create a (dangerous) script element, that can inject arbitrary code into the page.

```
var o = {toString: function( ) {
```

```
            o.toString = function(){return "script"};
            return "div";
          }
      };
dom.append(dom.tag(o).append(dom.text("alert('Hacked!')")));
```

The attack works because internally dom.tag has the structure

```
if (o != "script"){return document.createElement(o);}
```

and our object o is able to "lie" the first time it is converted to a string, claiming to be a "div" element. The call to document.createElement(o); has the authority to create any tag element that o can evaluate to. A safe alternative would be to limit such authority by construction, for example using an idiom like

```
var tag= {"div": "div", "script": "text"}[o];
return document.createElement(tag);
```

The third problem that we found is that the library function dom.text exposed to adverstisement code returns an object that leaks the authority to directly read DOM objects. From any DOM object, adverstisement code can get access to the enclosing document by accessing the "ownerDocument" property, as shown in the following example:

```
var a = dom.text(["hacked"]);
a[0].ownerDocument.location="http://attacker.com";
```

These examples do not indicate a fundamental flaw in the ADsafe design, but do suggest the need for a systematic formal analysis of isolation properties and the means to achieve them.

6.5 Related Work

In this section, we provide an overview of some related work on capability-based protection and its applications.

Capability-based protection in operating systems. Capability-based protection is a widely known method for operating-system-level protection, deployed in systems such as the Cambridge CAP Computer, the Hydra System, StarOS, IBM System/38, and the Intel iAPX423, all summarized in [42]. Another interesting operating system project is Amoeba [80], in which servers respond to messages sent with capabilities. Among prior operating system research, the closest to our work (to the best of our knowledge) is an analysis and proof of the EROS confinement mechanism [74], which uses an operational semantics of system execution. While there are some other similarities between their framework and our general setup, one substantial difference is that instead of defining authority as an over-approximation of heap actions that can be performed by a single object, they define authority for the whole system. Therefore, their main theorem shows that if a system performs an action, the system has authority to perform that action; this does not give precise information about whether a part of the system that performed the action was allowed to do so. Of course, the study of EROS involves system call actions, not the programming language actions studied in this dissertation.

Object-Capability model. Previous work on the object-capability model (e.g., [60, 58, 76]) outlines a number of principles and discusses advantages of the model. The principles and goals are presented using the *reference graph*, which is intended to both represent the references held by each object and also define the authority of an object as the aggregated authority of all reachable objects. While reference graphs are a pictorial and intuitive way to explaining how authority can be transferred from one object to another, we have not located technical work, similar to the present chapter, that connects authority analysis to operational semantics (or other semantics) of programming languages. Some recent work (e.g., [64]) defines precise forms of object-capability models in the context of process calculus.

Google Caja. The Google Caja framework [11] is a mechanism for sandboxing untrusted JavaScript, also based on the API+LBS architecture. In the Caja framework, the mashup host defines a *container* page that embeds the Caja libraries (API) and exposes them to untrusted mashup components written in the *Cajita* subset

of JavaScript (sandbox), which is designed to be a safe object-capability language. *Cajita* imposes significant restrictions on JavaScript — with statements, function declarations, for-in loops, this expressions, and expressions of the form o.p and o[e] are disallowed by Cajita. Moreover a Cajita module must be a function expression of the form function ($\tilde{y}$){s} with $\tilde{y}$ being the only variables that are free in s, and all object properties must be read or written only using the predefined Cajita functions readPub and setPub. In effect these restrictions ensure that a Cajita module derives all its authority *only* from the identifiers named in the module. The key idea is that the identifiers named in a module represent the capabilities held by it, and the resources reachable from the identifiers, in the heap graph made viewable by readPub and setPub, correspond to its authority. [49] formally proves that Cajita is capability-safe. Using the capability-based authority map of Cajita it is possible to design a sandboxing mechanism that enforces authority isolation and thus solves the inter-component isolation problem. However such a mechanism would be far more restrictive than our mechanism $\mathcal{S}^m_{\mathcal{G}}$, mainly due to the severe restrictions imposed by Cajita over JavaScript. On the other hand, since Cajita is designed to be a pure object-capability language, it enjoys many other properties such as *complete encapsulation* and *no ambient authority* (see [60, 58, 49] for a discussion).

Language-based information flow. Language-based research on information flow (see [72] for a survey) also has goals similar to the inter-component isolation goal considered in this work, but the enforcement techniques are significantly different. While our technique conservatively focusses on separating the access privileges held by two third-party components, information-flow techniques focus on analyzing the code executed by the components, both statically and at run-time, and then determining whether information can flow from one component to another. Furthermore, information-flow research also considers the much stronger goal of non-interference which requires the execution of one component to be completely oblivious of the execution of the other. The inter-component isolation goal considered in this work is weaker than non-interference, and does not forbid communication via timing and other covert channels. Some recent applications of information-flow to sandboxing mashup components are discussed in [51, 29].

Chapter 7

The Language SES

In Chapters 5 and 6, we described provable-correct sandboxing mechanisms for third-party ES3 code. We saw that even simple restrictions such as forbidding access to critical global variables are challenging to enforce on untrusted ES3 programs. Moreover, our research revealed multiple vulnerabilities in the enforcement of global variable isolation in two prominent ES3 sandboxing mechanisms: FBJS and ADsafe. The core underlying issue is that ES3 is not amenable to static analysis and has very poor support for defensive programming. Recognizing these difficulties, the ECMA Standards committee (TC39) developed a *strict mode* (ES5-strict) in the 5^{th} edition of the ECMAScript Standard [33]. ES5-strict imposes several restrictions on top of ES3, and thereby achieves lexical scoping and closure-based encapsulation. While ES5-strict is much more amenable to static analysis and defensive programming than ES3, two limitations still exist — (1) Mutability of built-in objects complicates defensive programming as one cannot rely on any built-in methods such as Object.prototype.toString, Function.prototype.call etc., and (2) Dynamic code generation using constructs such eval makes static analysis very challenging as some of the code that executes at run-time may not event exist statically.

In this chapter[1], we propose a sub-language SecureECMAScript (SES) of ES5-strict that supports static analysis and defensive programming by eliminating the aforementioned problems. Malicious use of built-in objects is restricted by making

[1]This chapter is based on joint work with Ulfar Erlingsson, Mark S. Miller and Jasvir Nagra.

all built-in objects except the global object immutable. Dynamic code execution is controlled by restricting eval and other dynamic code generation constructs. The language SES has been under proposal[2] by the ECMA committee (TC 39), for adoption within future versions of the JavaScript standard. While no current browser implements SES, it is possible to simulate the SES semantics in an ES5-strict environment, using an initialization script that effectively freezes all built-in objects and appropriately tames all dynamic code generation constructs. As discussed in Section 7.1, overall the language SES has a cleaner semantics and is more permissive compared to previous ES3 sandboxing sub-languages.

We define a small-step operational semantics and a formal analysis framework for the core of SES, called SES-light. The main difference between SES and SES-light is that SES supports getters/setters and SES-light does not. This is not a fundamental restriction and was introduced mainly to simplify the static analysis framework for SES-light that we developed subsequently. Also since none of the major JavaScript sandboxing mechanisms (FBJS, ADsafe and Caja) support getters/setters, this restriction should not be seen as a fundamental limitation by application developers either.

Organization. The rest of this chapter is organized as follows: Section 7.1 motivates the design of SES. In particular it states five key limitations of ES3 and discusses how the design of SES helps in overcoming them. Section 7.2 describes a small-step operational semantics for a core subset SES-light of SES. Section 7.3 formally proves that α-renaming of bound variables of SES-light programs is semantics preserving and finally Section 7.4 concludes.

7.1 From ES3 to ES5-strict to SES

In this section, we motivate the design of the language SES in three steps — we first describe five fundamental limitations of the ES3 language, then explain how the

[2]SES was originally conceived by Mark S. Miller who also led the proposal in the ECMA committee (TC 39). The first formal characterization of SES was developed by Ankur Taly and Mark S. Miller.

language ES5-strict gets rid of three of those limitations, and then finally explain how the language SES-light gets rid of all the limitations. We refer the reader to Chapter 2 for an overview of the ES3 and ES5-strict languages.

7.1.1 ES3 Limitations

ES3 suffers from five fundamental limitations that make static analysis and defensive programming challenging for the language.

(1) Lack of lexical scoping. As discussed in Chapter 5, ES3 does not support lexical scoping due to the presence of prototype chains on scope objects (or activation records) and the ability to place to first-class objects on the scope stack (using with). As a result, standard α-renaming of bound variables is unsound for ES3.

(2) Lack of closure-based encapsulation. Most browser implementations of ES3 provide a built-in property named "caller" in all function objects. On invoking a function, the "caller" property stores a reference to the calling function (as determined by the runtime call stack), or null if the invocation happened at the top level. The presence of the "caller" property on function objects completely breaks closure-baed encapsulation. To explain this further, we restate an example from Chapter 2. Consider a trusted function that takes an untrusted function as argument and checks possession of a secret before performing certain operations.

```
ES3> function trusted(untrusted, secret) {
       if (untrusted() === secret) {
         % process secretObj
       }
     }
```

Under standard programming intuition, this code should not leak secret to untrusted code. However the following definition of untrusted enables it to steal secret.

```
ES3> function untrusted() {return arguments.caller.arguments[1];}
```

The above function uses "caller" to access the other arguments passed to its caller. Since ES3 does not support private object properties, it is common for code to use

closures to simulate private fields. Lack of encapsulation in closures defeats this purpose.

(3) Ambient access to the global object. As discussed in Chapter 5, ES3 provides multiple ways for code to directly obtain a reference to the global scope — using the keyword this or invoking certain built-in methods of Object.prototype and Array.prototype in the global scope. Therefore in systems composed of trusted and untrusted code, unless specific restrictions, all global state remains untrusted. As a result defensive programming is very challenging.

(4) Mutable built-in objects. All built-in objects in ES3 are mutable, that is, their properties can be arbitrarily modified or deleted. Therefore in systems composed of trusted and untrusted code, untrusted code can modify properties of built-in objects to maliciously alter the behavior of trusted code. Mutability of built-in objects was the basis of the FBJS_{Nov09} hosting page isolation vulnerability described in Chapter 6. We found that a malicious Facebook application could modify the Function.prototype.apply method and thereby trick a trusted library function into granting it direct access to a DOM object.

(5) Dynamic code generation. The built-in eval function and Function constructor in ES3 allow new code to be generated dynamically at run-time. This is a huge hurdle for static analysis as some of the code that executes at run-time is not even visible statically.

7.1.2 From ES3 to ES5-strict

The restrictions imposed by ES5-strict over ES3, eliminate three out of the five ES3 limitations, namely *lack of lexical scoping*, *lack of closure-based encapsulation* and *ambient access to the global object*. The following table from Figure 2.1 presents the specific ES5-strict restrictions responsible for getting rid of each of the limitations.

Restriction	**Property enforced**
No delete on variable names	Lexical Scoping
No prototypes for scope objects	Lexical scoping
No with	Lexical scoping
No this coercion	No ambient access to global object
Safe built-in functions	No ambient access to global object
No "callee", "caller" properties on arguments objects	Closure-based encapsulation
No "caller", "arguments" on function objects	Closure-based encapsulation
No arguments and formal parameters aliasing	Closure-based encapsulation

We refer the reader to Chapter 2 for a discussion on how the above restrictions enforce the corresponding language properties.

7.1.3 From ES5-strict to SES

The restrictions imposed by SES on ES5-strict eliminate the two remaining limitations as well, namely, *mutable built-in objects* and *dynamic code generation* . SES addresses the first limitation by requiring that all built-in objects except the global object are completely frozen, that is, all properties are immutable and no properties can be added or deleted from them. While the global object is not frozen, all its built-in properties (like "Object", "Function", etc.) are made immutable.

The second limitation is addressed by restricting all dynamic code generation constructs so that the generate code does not have any free variables. (Unlike ES3, the free variables of a program are statically definable for SES.) We explain this restriction using the eval construct as an example. We defined a restricted form of eval, called *free-variable-restricted eval*, and denote it by eval_{nf}. As an example, the function declaration string "var x; x = 42;" can be safely evaluated as eval_{nf}("var x; x = 42;") since it does not have free variables. On the other hand evaluating the string "x = 42;" leads to an error as it includes free variables. This restriction makes it possible

to conservatively analyze programs containing $\mathsf{eval}_{\mathsf{nf}}$ calls as the code generated by $\mathsf{eval}_{\mathsf{nf}}$ is guaranteed to not access any non-local state. Other dynamic code generation constructs such as the Function constructor are restricted similarly.

For ease of programming SES also supports a special syntactic sugar written as $\hat{\mathsf{eval}}_{\mathsf{nf}}(m, m_1, \ldots, m_n)$, which dynamically converts the string m to code and executes it only if its free variables are contained in the set $\{m_1, \ldots, m_n\}$. As an example the string "var x = y;" can be safely evaluated as as $\hat{\mathsf{eval}}_{\mathsf{nf}}(\text{"var x = y"}, \text{"y"})$. The statement $\hat{\mathsf{eval}}_{\mathsf{nf}}(m, m_1, \ldots, m_n)$ is essentially syntactic sugar and can be expanded as $(\mathsf{eval}_{\mathsf{nf}}(\text{"function("} + m_1 + \text{","} + \ldots + \text{","} + m_n + \text{"){"} + m + \text{"}"}))(m_1, \ldots, m_n)$. We make use of $\hat{\mathsf{eval}}_{\mathsf{nf}}$ in Chapter 8 where we develop a mechanism for sandboxing SES code.

7.1.4 Implementing SES on an ES5-strict browser

The ideal deployment scenario for SES would be for browsers to primitively support it. Given the absence of such browsers, we present a first cut to an approach for emulating the SES restrictions on a browser supporting ES5-strict. The main idea is to run an initialization script that makes the heap compliant with the initial SES heap, and then use a static verifier on all code that runs subsequently. The goal of the static verifier is to ensure that the code is valid SES code.

The initialization script performs the following steps: (1) Freezes all built-in objects, except the global object, by invoking the built-in ES5-strict method Object.freeze on them, and (2) Replaces the built-in eval function and built-in Function constructor with a wrapper that uses an SES parser (written in ES5-strict) to ensure that dynamically generated code does not have any free variables.

We have an implementation [61] of the initialization script described above, but we do not have any rigorous proof of correctness for it yet. We conjecture that for all SES terms t, the execution of t on the initial SES heap and stack under the SES semantics, is safely emulated by the execution of t on the appropriately initialized ES5-strict heap and stack under the ES5-strict semantics.

7.2 An Operational Semantics for SES-light

In this section we describe a small-step operational semantics for a subset of SES called SES-light. The key difference between SES-light and SES is that SES-light does not support getters and setters. This is not a fundamental restriction and was applied mainly to simply the static analysis framework that we developed subsequently. Besides getters and setters, SES-light does not model certain other features of SES as well — in particular, the switch construct, for loops (we do model for in loops), built-in Function constructor (we do model a restricted form of eval), parsing (which is used at run time by the eval statement), and the built-in objects: Date, Math, Number and String. We skip these features as we believe that they do not add any new insights; if needed it would be straightforward to add them to the semantics.

The entire semantics of SES-light is approximately 27 pages long, formatted in ASCII, including a model for the built-in objects. SES-light is a sub-language of ES5-strict with frozen built-ins, with a restriction on eval, and without getters and setters. We therefore base the semantics of all expressions and statements, except $\mathsf{eval}_{\mathsf{nf}}$, on the ES5-strict specification. The semantics of the $\mathsf{eval}_{\mathsf{nf}}$ statement is modeled with the special free-variable restriction as discussed in Section 7.1.3.

While the semantics of the other statements are based on the ES5-strict specification, we do not completely mimic the internal structure followed by the specification. For example, we deviate from the specification in describing the grammar for expressions and statements. Similar to the ES3 specification, expressions in the ES5-strict specification are not side-effect free. This is however highly unconventional. Therefore in defining the syntax of SES-light, we model all side-effect causing expressions e as statements $y = e$. We also do not model the internal reference value $l^{*}m$ for describing the final value of an expression evaluation. Due to the side-effect free nature, all expressions in SES-light evaluate to a pure value or special exception values. Another deviation from the specification is that the syntax for property-lookup $e_1[e_2]$ can be optionally annotated with an annotation a, and can therefore be written as $e_1[e_2,\mathsf{a}]$. The annotation indicates a bound on the set of string values that expression e_2 can evaluate to. Analogous to the free-variable restriction enforced by $\mathsf{eval}_{\mathsf{nf}}$, this

annotation also enforces static restrictions on dynamically generated content and thus benefits static analysis.

Given the volume of the entire semantics, we only describe the main semantic functions and some representative axioms and rules here; the full semantics is currently available online [79]. We begin with the syntax of SES-light.

7.2.1 Syntax

The syntax for all top-level (user-level) SES-light programs is given in figures 7.1 and 7.2. It is divided into values, expressions and statements. Similar to the grammar for ES3, we follow systematic conventions about the syntactic categories of metavariables, to give as much information as possible about the intended type of each operation. In the grammar, UN, BIN range respectively over unary and binary operators. We abbreviate $t_1, \ldots, t_n$ with $\tilde{t}$ and $t_1 \ldots t_n$ with $t*$ ($t+$ in the nonempty case). $[t]$ means that t is optional, $t \mid s$ means either t or s. In case of ambiguity we escape with apices, as in "[" t "]".

A value (v) is either a primitive value (pv) or a heap location (l). Similar to ES3, primitive values are standard with two special values @undefined and @NaN. Heaps locations are prefixed by the symbol #. They include certain constant heap locations #global, #obj, ..., etc., which correspond to built-in objects, and the special location @null. Fresh heap locations range over #1, ... Variables are either strings names foo, bar, ... or special internal names $@_1, \ldots$. We explain the purpose of these internal names later. Exceptions (w) in SES-light consist of two special values TypeError or RefError.

Expressions are either variables or values. Statements include assignment, property load, property store, and all representative control flow constructs from ES5S. The statement $\mathsf{eval}_{\mathsf{nf}}(e)$ is the special free-variable-restricted eval statement. All statements are written out in a normal form, similar to the A-Normal form of featherweight Java [6]. It is easy to see that using temporary variables, all complex statements from ES5S, except setters/getters and $\mathsf{eval}_{\mathsf{nf}}$, can be re-written into semantics-preserving normalized statements. For example, y = o.f.g.h() can be re-written to

Variables and Values

v ::=	$l \mid pv$	values
w ::=	TypeError | RefError	exceptions
l ::=	#global | #Object | ...	locations
	@null | #1 | ...	
pv ::=	$m \mid n \mid b \mid$ @null | @undefined	primitive values
m ::=	"foo" | "bar" | ...	strings
n ::=	$-n \mid$ @NaN | @Infinity | 0 | 1 | ...	numbers
b ::=	true | false	booleans
fv ::=	function $x(\tilde{y})\{s\}$	function values
an ::=	$All | $Num | ...	annotations
x, y ::=	this | foo | bar | ...	user-variables
	$@_1$ | ...	internal-variables

Figure 7.1: Syntax for SES-light variables and values

$a=o.f ; $b=$a.g ; y=$b.h() with temporary variables $a and $b.

The syntax for property-lookup is augmented with property annotations. Examples of annotations are: $Num which represents the set {"0", "1", ...}, $Builtin which represents the sets of built-in method names {"toString", "valueOf", ...}, etc. We use the annotation $All to represent the domain of all strings. Using $All, we can trivially translate an un-annotated property lookup to annotated property lookup. We denote the set of all annotations by $\mathcal{A}$ and assume a map $Ann : Str \rightarrow 2^{\mathcal{A}}$ specifying the valid annotations for a given string.

Since the semantics is small step, it also introduces new statements in the program state for book-keeping. Such statements are called *internal statements* and are prepended with the symbol "@" in order to distinguish them from the user-level statements. In the next few sections, we will elaborate on a few of the internal statements as we describe the semantic rules. The full semantics, defined in [45], gives the entire grammar for internal statements as well.

Expressions and Statements:

$e ::=$	$x \mid v$	
$s, t ::=$	$y = e$	expression statement
	$y{=}e_1\ BIN\ e_2$	binary expression
	$y{=}UN\ e$	unary expression
	$y{=}e_1\text{“[”}\,e_2, an\text{“]”}$	load
	$e_1\text{“[”}\,e_2, a\text{“]”} = e_3$	store
	$y{=}\{[p\tilde{n}{:}e]\}$	object literal
	$y{=}\text{“[”}\,[\tilde{e}]\,\text{“]”}$	array literal
	$y{=}e_1([\tilde{e_2}])$	call
	$y{=}e_1\text{“[”}\,e_2, a\text{“]”}(\tilde{e_3})$	invoke
	$y{=}\textsf{new}\ e_1([\tilde{e_2}])$	new
	$y{=}\textsf{function}\ [x]\ ([\tilde{z}])\{s\}$	function expression
	$\textsf{function}\ x\ ([\tilde{z}])\{s\}$	func decl
	$\textsf{eval}(e)$	eval
	$\textsf{return}\ e$	return
	$\textsf{var}\ x$	var
	$\textsf{throw}\ e$	throw
	$s; t$	sequence
	$\textsf{if}\ (e)\ \textsf{then}\ s\ [\textsf{else}\ t]$	if
	$\textsf{while}\ (e)\ s$	while
	$\textsf{for}\ (x\ \textsf{in}\ e)\ s$	forin
	$\textsf{try}\ \{s\}\ [\textsf{catch}\ (\textsf{x})\ \{s_1\}]\ [\textsf{finally}\ \{s_2\}]$	try
$pn ::=$	$m \mid n \mid x$	property names

Figure 7.2: Syntax for SES-light expressions and statements

Props $p ::=$	m | @extensible | @class | @code | @prototype | $@_1$ | ...	properties
$a ::=$	writable | configurable | enumerable	attributes
$cl ::=$	fv, A	closures
ObjVals $ov ::=$	$v\{[\tilde{a}]\} \mid cl$	object values
RVal $rv ::=$	$v\{[\tilde{a}]\} \mid \bot$	record values
$o ::=$	$Props \rightharpoonup ObjVals$	objects
$R ::=$	$Vars \rightharpoonup RVal$	records
$H, K ::=$	$\tilde{l{:}o}$	heaps
$A, B ::=$	$A{:}R \mid ERG$	stacks

Figure 7.3: Heaps and Stacks in SES-light

7.2.2 Building Blocks

Heaps and Stacks. The complete definitions of Heaps and Stacks are present in Figure 7.3. Similar to ES3, heaps (H) in SES-light map heap locations (l) to objects (o). Objects are partial maps from properties (p) to object values. Properties are either strings or internal properties such as @prototype, @class, etc. Object values are either function closures or regular values with certain optional attributes (a). Attributes indicate certain restrictions on property access. A closure is a pair of a function value (fv) and an execution stack which we describe next.

Unlike ES3, execution stacks in ES5-strict (and therefore SES-light) consist of standard variable records rather than objects. Variable records (R) are partial maps from variables to record values. A record value is either a regular value with certain optional attributes (a), or the special value $\bot$ denoting an uninitialized variable name. The base of an execution stack is always a special record called ERG which denotes the global scope. In order to distinguish form ES3 stacks, we use the symbols A, B to range over SES-light stacks. We use $Heaps^{\text{SESl}}$ and $Stacks^{\text{SESl}}$ as the universe of all possible SES-light heaps and stacks respectively. For ease of presentation, unless needed we always elide the attribute part of object values and record values.

Helper Functions. For a partial map f, $dom(f)$ is the set of elements on which it

is defined. Therefore for a heap H, $dom(H)$ gives the set of allocated heap locations. For a value v and partial map f, $f[x \rightarrow v]$ denotes the map obtained by updating the value of $f(x)$ to v. We use maps $sToi$ and $iTos$ to convert between strings and identifiers and vice versa. As an example, $sToi(\text{“a”}) = \mathsf{a}$ and $iTos(\text{“a”}) = \mathsf{a}$

Types. Similar to ES3, values in SES-light are dynamically typed. The internal types are:

$$T ::= \quad \mathsf{Undefined} \mid \mathsf{Null} \mid \mathsf{Boolean} \mid \mathsf{String} \mid \mathsf{Number} \mid \mathsf{Object}$$

Types are used to determine conditions under which certain semantic rules can be evaluated. Given a value v, we assume a function $Type(v)$ that returns the internal type of value v.

Semantic Functions. Unlike ES3, expressions in SES-light are side-effect free. Their semantics is therefore described using a function.

$$[\![e]\!] : Heaps^{\mathrm{SESl}} \times Stacks^{\mathrm{SESl}} \rightarrow VErrs$$

Here $VErrs$ is the set of all SES-light values (v) and exceptions (w).

The semantics of statements are described using a set of state reduction rules which are formally denoted by $(\Sigma, Rules^{\mathrm{SESl}})$. Σ is the universe of program states which are triples H, A, t consisting of a heap H, stack A and statement t. The general form of a state reduction rule is $\dfrac{\langle premise \rangle}{H, A, t \rightarrow K, B, s}$. As in the case of ES3, the reduction rules are divided in transition axioms and contextual rules. The evaluation of a statement terminates with a *termination value* tv which is defined as follows:

$$tv ::= \quad \mathsf{N} \mid \mathsf{Ret(v)} \mid \mathsf{Th(vw)}$$

The value N denotes normal completion of execution, $Ret(v)$ denotes function return and $Th(v)$ denotes disrupted execution. For the latter two, v denotes the value returned and thrown respectively.

7.2.3 Expression Semantics

In order to describe the semantics of expressions we first describe the semantics of property and variable lookup in SES-light.

Property Lookup and Identifier Lookup. Property and identifier lookup in SES-light are both defined as functions over a heap and stack. Property lookup uses the prototype-based inheritance mechanism. Similar to ES3, the prototype chain in SES-light is modeled by having an internal property @Prototype in each object that stores a reference to its prototype object. Given a heap H, the value of property m for object at location l is given by the function $Prototype(H, l, m)$, defined as follows:

$$\frac{m \notin dom(H(l)) \quad H(l)(\texttt{@prototype}) = l_1}{Prototype(H, l, m) = Prototype(H, l_1, m)}$$

$$\frac{m \in dom(H(l)) \quad v = H(l)(m)}{Prototype(H, l, m) = v} \qquad Prototype(H, \texttt{@null}, p) = \texttt{@undefined}$$

Identifier lookup for SES-light is defined in the standard way by traversing down the stack of variable records. If the bottommost record ERG is reached then the corresponding property name is looked up in the global object, and if the lookup fails then a RefError is thrown. It is formalized using the function $Lookup(H, A, x)$, that internally makes use of the predicate $HasProperty(H, l, m)$ which checks if property m appears anywhere on the prototype chain of object $H(l)$.

$$\frac{x \in R \quad v = R(x)}{Lookup(H, A : R, x) = v} \qquad \frac{\neg HasProperty(H, \#global, x)}{Lookup(H, ERG, x) = \texttt{RefError}}$$

$$\frac{x \notin dom(R)}{Lookup(H, A : R, x) = Lookup(H, A, x)}$$

$$\frac{HasProperty(H, \#global, x)}{Lookup(H, ERG, x) = Prototype(H, \#global, x)}$$

Using the *Lookup* function, the expression semantics map $[\![e]\!] : Heaps \times Stacks \to Vals$ is defined as:

$$[\![x]\!]HA = Lookup(H, A, x) \qquad [\![v]\!]HA = v$$

7.2.4 Statement Semantics

As discussed earlier, semantics of statements are expressed using small-step state transition rules of the form $\frac{\langle premise \rangle}{H, A, t \to K, B, s}$. Similar to the semantics of ES3, there

are two kinds of reduction rules: transition axioms and contextual rules. One structural difference with the semantics of ES3 is in the use of internal variables $@_1, \ldots$ for storing intermediate results, instead of using internal contexts. Since SES-light is lexically scoped it is possible to use variables for storing results, that then get read by subsequent statements. For example, the following rule expresses the semantics of the object literal statement $\{pn_1{:}e_1, \ldots, pn_m : e_m\}$ using the internal variable $@_1$.

$$\frac{@ \in \mathit{freshVar}()}{H, A, \{pn_1{:}e_1, \ldots, pn_m : e_m\} \rightarrow H, A, @ = \{\}; @.pn_1 = e_1, \ldots, @.pn_m = e_m}$$

Above, $\mathit{freshVar}()$ returns a fresh @-variable. The namespace of @-variables is isolated from user-level code.

Contextual Rules. There are two main contextual rules for statements. The first one is the contextual rule for evaluating sub-expressions.

$$\frac{[\![e]\!]HA = v \quad \neg \mathit{IsError}(v)}{H, A, sCe[e] \rightarrow H, A, sCe[v]} \qquad \frac{[\![e]\!]HA = v \quad \mathit{IsError}(v)}{H, A, sCe[e] \rightarrow H, A, \mathsf{Th}(v)}$$

Here sCe is a statement context for evaluating expressions, and $\mathit{IsError}(v)$ is a predicate for checking whether v is one of the error values TypeError or RefError. Examples of sCe contexts are $y = \_\ \mathit{BIN}\ e$, $y = v\ \mathit{BIN}\ \_$, and so on (see [79] for the complete list). The second main contextual rule is the one for evaluating sub-statements.

$$\frac{H, A, s \rightarrow K, B, t}{H, A, sCs[s] \rightarrow K, B, sCs[t]}$$

Here sCs is a statement context for evaluating statements. Examples of such contexts include $\_\ ;\ s$, try $\_$ finally s, ... and so on (see [79] for the complete list).

Assignments. The first step in the evaluation of an assignment statement $y = e$ is to evaluate the expression e to a value. This step is modeled by making the context $y = \_$, a statement context for evaluating sub-expressions. Once the expression e evaluates to a value v, the evaluation of the statement $y = v$ is described by the following axioms:

$$\frac{\neg \mathit{CanPutVar}(A, y)}{H, A, y = v \rightarrow H, A, \mathsf{Th}(\mathsf{TypeError})}$$

$$\frac{\mathit{CanPutVar}(A, y) \qquad K, B = \mathit{Update}(H, A, y, v, \{\mathsf{writable}\})}{H, A, y = v \rightarrow K, B, \mathsf{N}}$$

Above, $\mathit{CanPutVar}(A, y)$ holds if the variable y has an attribute writable in the variable record to which it resolves. $\mathit{Update}(H, A, y, v, \{\tilde{a}\})$ returns a new heap and stack K, B after updating the value of variable y to the value v with attributes $\{\tilde{a}\}$. Note that the heap can also potentially change in an update operation if the variable resolves to the global variable-record ERG. In such a case the corresponding property of the global object gets updated.

Implicit type conversions. Similar to ES3, values in SES-light are dynamically typed and get coerced implicitly depending on where they appear. We model such implicit type conversions by internal statements @TS(y,v), @TN(y,v), @TB(y,v), @TO(y,v) which assign to y after converting v to a string, number, boolean and object respectively. We describe the rules for @TS(y,v) as an example.

$$\frac{@ = \mathit{freshVar}()}{H, A, @\mathsf{TS}(y,l) \rightarrow H, A, @\mathsf{TP}(@,l,\text{“toString”}); @\mathsf{TS\text{-}help}(y,@)}$$

$$\frac{m = \mathit{Prim2Str}(pv)}{H, A, @\mathsf{TS\text{-}help}(y,pv) \rightarrow H, A, y = m}$$

Above, @TS(y,l) allocates a fresh internal variable @ and invokes @TP(@,l,“toString”) which converts l to a primitive value and assigns it to the internal variable @. The optional argument “toString” specifies that the primitive conversion should first invoke the “toString” method on l. (In the absence of this argument the method invoked by default is “valueOf”.) Next, the internal statement @TS-help(y,@) converts the primitive value stored in @ to a string and stores it in y.

Load. The first step in the evaluation of an annotated load statement $y = e_1[e_2, an]$ is to evaluate the expressions e_1 and e_2 to values. This is modeled by making the contexts $y = \_[e]$, $y = v[\_]$ statement contexts for evaluating expressions. Once expressions e_1, e_2 evaluate to values v_1, v_2 respectively, the remaining steps are modeled using the following transition axioms.

$$\frac{@_1, @_2 = freshVar()}{H, A, y{=}\ v_1[v_2,an] \rightarrow H, A, \texttt{@TO}(@_1,v_1);\texttt{@TS}(@_2,v_2);\ \texttt{y} = @_1[@_2,\texttt{an}]} \quad (1)$$

$$\frac{an \notin Ann(m)}{H, A, y{=}\ l[m,an] \rightarrow H, A, \texttt{Th(TypeError)}} \quad (2)$$

$$\frac{v = Prototype(H, l, m) \qquad an \in Ann(m)}{H, A, y{=}\ l[m,an] \rightarrow H, A, y{=}\ v} \quad (3)$$

Above, Rule 1 generates internal statements $\texttt{@TO}(@_1,v_1)$ and $\texttt{@TS}(@_2,v_2)$ for converting v_1 to an object and v_2 to a string. Here $@_1$ and $@_2$ are fresh internal variables used to store the results of the conversions. Rule 2 checks if the string obtained from v_2 does not satisfy the annotation an and throws a TypeError. If the string does satisfy the annotation then Rule 3 carries out the corresponding property access and assigns the value obtained to variable y.

(Free-variable-restricted) Eval. In order to describe the semantics of the free-variable-restricted eval statement, we first precisely define the notion of free variables for a statement. Given a user statement s, the free and bound variables of the statement, denoted respectively by $Free(s)$ and $BV(s)$, are formally defined in Figures 7.4 and 7.5. The bound variables are the union of all identifiers appearing in top-level variable and functions declarations in s. Here top-level declarations are those that are nested in s but not within any inner function declarations. The free variables are essentially the union of the following: (i) all top-level identifiers that are not bound variables of s and are not named by any enclosing catch blocks, and (ii) free variables contained in the bodies of all top-level functions declarations, that are not bound variables of s.

We now present the semantics for the eval statement $\texttt{eval}_{\texttt{nf}}(e)$. The first step is to evaluate the argument e which is handled by the contextual rule for evaluating expressions. The result obtained is then converted to a string m using the internal statement @TS. The semantics of evaluating the string m are then modeled by the following rules.

$$\frac{Parse(m) = \mathsf{TypeError}}{H, A, \mathsf{eval_{nf}}(m) \rightarrow H, A, Th(\mathsf{TypeError})} \quad (1)$$

$$\frac{Parse(m) = s \quad Free(s) \neq \emptyset}{H, A, \mathsf{eval_{nf}}(m) \rightarrow H, A, Th(\mathsf{TypeError})} \quad (2)$$

$$\frac{\begin{array}{c} Parse(m) = s \qquad Free(s) = \emptyset \qquad R = NewRec() \\ H_1, R_1 = FD(H, R, s) \qquad K, R_2 = VD(H_1, R_1, s) \end{array}}{H, A, \mathsf{eval_{nf}}(m) \rightarrow K, A : R, \mathsf{@EVAL(s,}A\mathsf{)}} \quad (3)$$

The first step is to parse the argument string m. If the parsing fails (Rule 1) then a TypeError exception is thrown. If the parsing succeeds then the free variables of the parsed statement s are computed using the function *Free* as defined in Figures 7.4 and 7.5. (We discuss the definition of *Free* later in this Section 7.3.). If the set of free variables is non-empty (Rule 2) then a TypeError exception is thrown. Otherwise, a new activation record is created and initialized with all the variable and function declarations occurring in statement s. This step is formalized by functions *VD* and *FD* which take a heap, a variable record and a statement, and initialize them with all the variable and functions declarations respectively. After this step, control is transferred to a special @EVAL statement that evaluates the statement s (using contextual rules) and then restores the original stack A in case of normal termination. An exception is thrown in case of an abnormal termination. The key transition axioms for the @EVAL statement are shown below

$$H, A, \mathsf{@EVAL(N,B)} \rightarrow H, B, \mathsf{N}$$

$$H, A, \mathsf{@EVAL(Ret,B)} \rightarrow H, B, \mathsf{Th(TypeError)}$$

$$H, A, \mathsf{@EVAL(Th(}vw\mathsf{),B)} \rightarrow H, B, \mathsf{Th(}vw\mathsf{)}$$

As discussed earlier, for ease of programming SES-light also supports a special eval statement $\hat{\mathsf{eval}}_{\mathsf{nf}}(m_0, \tilde{m})$ which is syntactic sugar for the statement

$$(\mathsf{eval_{nf}}(\text{“function(”} + sTo\tilde{i}(m) + \text{“)\{”} + m_0 + \text{”\}”}))(sTo\tilde{i}(m))$$

where $sToi(m)$ denotes the identifier corresponding to string m. From the definition of free variables of a term, it follows that a statement $\hat{\mathsf{eval}}_{\mathsf{nf}}(m_0,\tilde{m})$ evaluates normally if the string m parses to a valid statement whose free variables are contained in the set $\{\tilde{sToi}(m)\}$. Therefore the effective evaluation rule for $\hat{\mathsf{eval}}_{\mathsf{nf}}(m_0,\tilde{m})$ is

$$\frac{\begin{array}{lll} Parse(m_0) = s & Free(s) \subseteq \{\tilde{sToi}(m)\} & R = NewRec() \\ H_1, R_1 = FD(H, R, s) & K, R_2 = VD(H_1, R_1, s) & \end{array}}{H, A, \hat{\mathsf{eval}}_{\mathsf{nf}}(m_0,\tilde{m}) \rightarrow K, A : R, \mathsf{@EVAL(s,}A\mathsf{)}}$$

All analyses for SES-light described in this dissertation only use $\hat{\mathsf{eval}}_{\mathsf{nf}}$ and not $\mathsf{eval}_{\mathsf{nf}}$.

Built-in Objects. The ES5-strict specification defines a set of objects and functions that must be initially present in all execution environments. We model these by defining an initial heap H_0^{SESl} and stack A_0^{SESl} that are populated with the predefined objects. The initial stack A_0^{SESl} is essentially a stack consisting of only the special global variable record ERG. For ease of analysis, we only model a small set of the built-in objects on the initial heap H_0^{SESl}, namely, the global object, constructors Object, Array, methods "toString", "valueOf", "hasOwnProperty", "propertyIsEnumerable" of Object.prototype, methods "toString", "call", "apply" of Function.prototype and methods "toString", "join", "concat", "push" of Array.prototype. The reduction rules for the aforementioned methods are very similar to those defined for the built-in methods in ES3, and can be found online [79]. We limit ourselves to such a small set of built-in objects solely for simplifying the analysis framework that we develop for analyzing SES-light programs[3]. It is completely straightforward to extend the semantics so that it includes a model for all the built-in objects.

As mentioned in Section 7.1.3, SES-light imposes the restriction that all built-in objects, except the global object, are *transitively immutable*, which means their @extensible property is set to false and none of their properties have the attributes configurable or writable. Furthermore, none of the built-in properties of the global object have the attributes configurable or writable.

[3]We acknowledge that as a result the analysis framework only applies to programs that do not access any built-in objects outside this set.

$Free : Terms^{\mathrm{SESl}}_{user} \to 2^{Vars}$

$FV : Terms^{\mathrm{SESl}}_{user} \times 2^{Vars} \to 2^{Vars}$

$BV : Terms^{\mathrm{SESl}}_{user} \to 2^{Vars}$

Given a user-statement s, $Free(s)$ is defined as $FV(s, BV(s))$

Given a user-statement s and a set $\mathcal{B}$ of user-variables, $BV(s)$ and $FV(s, \mathcal{B})$ are defined below.

Let $V(\tilde{e}) := \{\tilde{e}\} \cap Vars$.

s	$BV(\mathbf{s})$	$\mathbf{FV}(\mathbf{s}, \mathcal{B})$
y= e	$\emptyset$	$V(y, e) \setminus \mathcal{B}$
y= e_1 *BIN* e_2	$\emptyset$	$V(y, e_1, e_2) \setminus \mathcal{B}$
y= *UN* e	$\emptyset$	$V(y, e) \setminus \mathcal{B}$
y= e_1[e_2,a]	$\emptyset$	$V(y, e_1, e_2) \setminus \mathcal{B}$
e_1[e_2,an] = e_3	$\emptyset$	$V(y, e_1, e_2, e_3) \setminus \mathcal{B}$
y= $\{x : e\}$	$\emptyset$	$V(y, \tilde{e}) \setminus \mathcal{B}$
y= $[\tilde{e}]$	$\emptyset$	$V(y, \tilde{e}) \setminus \mathcal{B}$
y= $e(\tilde{e_i})$	$\emptyset$	$V(y, e, \tilde{e}) \setminus \mathcal{B}$
y= e[e',an]$(\tilde{e_i})$	$\emptyset$	$V(y, e, e', \tilde{e}) \setminus \mathcal{B}$
y= new e$(\tilde{e_i})$	$\emptyset$	$V(y, e, \tilde{e}) \setminus \mathcal{B}$
eval$_{\mathsf{nf}}$(e)	$\emptyset$	$V(e) \setminus \mathcal{B}$
return e	$\emptyset$	$V(e) \setminus \mathcal{B}$
var x	$\{x\}$	$V(x) \setminus \mathcal{B}$
throw e	$\emptyset$	$V(e) \setminus \mathcal{B}$
y= function $x(\tilde{z})\{s_1\}$	$\{x\}$	$FV(s_1, \mathcal{B} \cup \{\tilde{z}\} \cup BV(s_1) \cup V(y, x) \setminus \mathcal{B}$
function $x(\tilde{z})\{s_1\}$	$\{x\}$	$FV(s_1, \mathcal{B} \cup \{\tilde{z}\} \cup BV(s_1) \cup V(x) \setminus \mathcal{B}$

Figure 7.4: Free variables of SES-light terms (Part 1)

$\mathbf{s}$	$BV(\mathbf{s})$	$\mathbf{FV}(\mathbf{s}, \mathcal{B})$
s_1;s_2	$BV(s_1) \cup BV(s_2)$	$FV(s, \mathcal{B}) \ \cup \ FV(t, \mathcal{B})$
if (e) then s_1 else s_2	$BV(s_1) \cup BV(s_2)$	$(V(e) \setminus \mathcal{B}) \cup FV(s, \mathcal{B}) \ \cup \ FV(t, \mathcal{B})$
while (e) s_1	$BV(s_1)$	$(V(e) \setminus \mathcal{B}) \ \cup \ FV(s, \mathcal{B})$
for $(x$ in $e)$ s_1	$BV(s_1)$	$(V(e) \setminus \mathcal{B}) \ \cup \ FV(s, \mathcal{B} \cup \{x\})$
try$\{s_1\}$catch$(x)\{s_2\}$ finally$\{s_3\}$	$BV(s_1) \cup BV(s_2)$ $\cup \ BV(s_3)$	$FV(s_1, \mathcal{B}) \ \cup \ FV(s_3, \mathcal{B})$ $\cup \ (FV(s_2, \mathcal{B} \cup \{x\}))$

Figure 7.5: Free variables of SES-light terms (Part 2)

$Wf_h(H)$ holds for a heap H is iff the following conditions are true.

- *Every allocated object has the internal properties @class and @prototype*

$$\forall l \in dom(H) : \{\text{@class}, \text{@prototype}\} \subset props(H(l))$$

- *Every allocated Function object has the properties @closure and "prototype", and the function values stored in the @closure property are well formed. $\forall l \in dom(H(l)$:*

$$H(l).\text{@class} = \text{"Function"} \Rightarrow \left(\begin{array}{l} \{\text{@closure}, \text{"prototype"}\} \subseteq props(H(l)) \\ \forall fv, A : H(l).\text{@closure} = fv, A \Rightarrow Wf_t(fv) \end{array} \right)$$

- *Every allocated Array object has a "length" property whose value is of type Number.*

$$\forall l \in dom(H(l) : H(l).\text{@class} = \text{"Array"} \Rightarrow Type(H(l).\text{"length"}) = \text{"Number"}$$

- *All internal properties of built-in objects have the same value as as that on the heap H_0^{ES3}.*

$$\forall l \in Builtins, p : p \in prop^{int}(H_0^{SESl}(l)) \Rightarrow H(l).p = H_0^{SESl}(l).p$$

- *For all internal statements @TS-help(x, e) and TN-help(x, e), expression e is a variable or a primitive value.*
- *The prototype chain for any object never contains a cycle.*

Figure 7.6: Well-formedness of SES-light heaps

7.3 Analysis Framework

In this section, we set up a basic framework for analyzing SES-light programs. Unless stated otherwise, all notations and definitions apply only to the semantics of SES-light. In order to avoid additional notational overhead, we borrow some notations from the analysis framework for $ES3$.

7.3.1 Notations and Definition

We denote by *Locs*, *Vals*, *Props*, *Vars*, *Stmts* the set of all possible SES-light locations, values, properties, variables and statements as defined in figures 7.1 and 7.2. We split *Props* and *Vars* into user and internal parts by defining $Props := Props_{user} \uplus Props_{int}$, $Vars := Vars \uplus Vars$. The set of SES-light statements is split as $Terms := Terms^{\text{SESl}}_{user} \uplus Terms^{\text{SESl}}_{int}$ where $Terms^{\text{SESl}}_{user}$ and $Terms^{\text{SESl}}_{int}$ are the set of user and internal statements respectively.

Recall from Section 7.2.2 that a heap is a map from heap locations to objects, an object is a record from properties to pure values or closures, and a stack is a list of activation records. For a heap location l and a term t and a stack A, we say $l \in t$ iff l occurs in t. Given a heap H, we define $dom(H)$ as the set of all allocated heap locations of H, and given an object o, we define $props(o)$ as the set of all properties of o. We split $props(o)$ as $props(o) := prop^{user}(o) \uplus prop^{int}(o)$ where $prop^{user}(o)$ and $prop^{int}(o)$ are the sets of all user and internal properties of object o. H_0^{SESl} is the initial heap containing all the built-in objects and $A_0^{\text{SESl}} := ERG$ is the initial stack containing of only the global activation record. We use $Builtins := dom(H_0^{\text{SESl}})$ to denote the set of heap locations of all the built-in objects.

Given a state $S := (H, A, t)$, $heap(S)$, $stack(S)$ and $term(S)$ denote the heap, stack and term part of the state. We note that by design, the term part of a state is always a statement. Given states S and T, we say S reaches T in many steps, denoted by $S \rightsquigarrow T$, iff either $S \to T$ holds or there exists states $S_1, \ldots, S_n$ $(n \geq 1)$ such that $S \to S_1 \to \ldots \to S_n \to T$ holds. For a state S, $\tau(S)$ is the possibly infinite sequence of states $S, S_1, \ldots, S_n, \ldots$ such that $S \to S_1 \to \ldots \to S_n \to \ldots$. Given a set of states $\mathcal{S}$, $Reach(\mathcal{S})$ is the set of all reachable states, defined as $\{S' \mid \exists S \in \mathcal{S} : S \rightsquigarrow S'\}$. Finally

a state S is *initial* if $heap(S) = H_0^{\text{SESl}}$, $stack(S) = A_0^{\text{SESl}}$ and $term(S) \in Stmts_{user}^{\text{SESl}}$.

7.3.2 Labelled Semantics

We augment the semantics of SES-light with *labels*, that provide a tracking mechanism for the heap locations and environment records allocated during the execution of a term. Intuitively, labels are attached to all nodes in the syntax tree of a term. For example, the statement if (x) then y = {a:42} else y = 1 is labelled as $\hat{l}_1$:if (x) then $\hat{l}_2$: y = {a:42} else $\hat{l}_3$:y=1 using the labels $\hat{l}_1, \hat{l}_2, \hat{l}_3$. In general, we use $\mathcal{L}$ as the universe of all labels.

Labels are also attached to heap locations and stack frames, based on the term whose evaluation created them. Each rule $\dfrac{\langle premise \rangle}{H, A, t \rightarrow K, B, s}$ is augmented so that all dynamically generated sub-terms of s, all allocated locations and all allocated activation record, carry the label of term t. We give the modified rule for object allocation as an example

$$\frac{l = freshLoc(loc) \qquad K = H[(\hat{l} : l) \rightarrow NewObject(\#objproto)]}{H, A, \hat{l} : y = \{\} \rightarrow K, A, \hat{l} : y = l}$$

Finally, unique labels are attached to all locations on the initial heap and stack H_0^{SESl}, A_0^{SESl}. We use $\hat{l}_g$ as the label for the global object. From here onwards, we will only consider the labelled semantics for SES-light. To avoid notational overhead, we will use the same symbols l, R and s for labelled locations, activation records and statements and define $Lab(l)$, $Lab(R)$ and $Lab(s)$ respectively as the labels associated with them.

7.3.3 Well-formedness

We define the notion of *well-formedness* for SES-light states. A SES-light state $S = (H, A, t)$ is well formed if the heap H is well formed, and the stack A and term t are well formed with respect to the heap H. Therefore $Wf(S) := Wf_h(H) \wedge Wf_s(A, H) \wedge Wf_t(t, H)$. The definition of $Wf_h(H)$ for a heap H is given in Figure 7.6. $Wf_s(A, H)$ holds for a stack A iff $\forall l : l \in A \Rightarrow l \in dom(H)$. $Wf_t(t, H)$ holds for a term t iff

$t \in \mathit{Stmts}$ and $\forall l : l \in t \Rightarrow l \in \mathit{dom}(H)$. We prove a progress and preservation theorem, showing that evaluation of a well-formed state never gets stuck (Progress) and that well-formedness of states is preserved across evaluation (Preservation).

Theorem 8 *For all states S_1 such that $Wf(S_1)$ holds:*

- *(Preservation) If there exists a state S_2 such that $S_1 \rightarrow S_2$, then $Wf(S_2)$ holds.*
- *(Progress) If $term(S_1) \notin \{N\} \cup \{Th(v) \mid v \in \mathit{Vals}\}$ then there exists a state S_2 such that $S_1 \rightarrow S_2$.*

The proof of the above theorem is carried out using an induction on the set of reduction rules for the preservation part, and a structural induction on the terms for the progress part. Due to the sheer volume of the semantics, we only describe a sketch of the proof in the appendix (Section A.4).

7.3.4 α-Renaming

As discussed earlier, SES-light is a lexically scoped language. We formalize this property by defining a semantics preserving procedure for renaming bound variables in a SES-light statement. The procedure is parametric on a variable renaming map $\alpha : \mathit{Vars} \times \mathcal{L} \rightarrow \mathit{Vars}$ that generates unique names for a particular scope label. In order to define the procedure we first define the concept of *closest bounding label* of an identifier.

Given a labelled user statement s, the *closest bounding label* of an identifier x appearing in s is defined as the label of the closest enclosing *function expression*, *function declaration* or try-catch-finally statement that has x as one of its bound variables.

Definition 19 *[α-Renaming] Given a labelled user statement s and a variable renaming map $\alpha : \mathit{Vars} \times \mathcal{L} \rightarrow \mathit{Vars}$, the renamed statement $Rn(s, \alpha)$ is obtained by replacing each variable x appearing in s, such that $x \notin \mathit{Free}(s)$, with $\alpha(x, \hat{l})$ where $\hat{l}$ is the closest bounding label of x.*

Next, we state and prove our main result which is that the above procedure is semantics preserving. In order to prove this result, we extend the renaming function Rn

to labelled program traces and show that renamed and unrenamed traces are *bisimilar*. Renaming is first extended to States by individually renaming the heap, stack and term components. A heap is renamed by appropriately renaming all closures appearing on it and a stack is renamed by renaming all variables using the label of the property record in which it appears. We refer the reader to Appendix A.4 for a precise definition of state renaming.

Theorem 9 *For all states S, $Rn(\tau(S)) = \tau(Rn(S))$.*

The proof of the above theorem is carried by an induction on the set of reduction rules. Due to the sheer volume of the semantics, we only describe a sketch of the proof in the appendix (Section A.4).

7.4 Summary

In this chapter, we proposed a sub-language SES of ES5-strict that is lexically scoped, and is amenable to static analysis and defensive programming. We developed a small-step operational semantics for a core fragment of SES, namely SES-light, and formally showed that it supports semantics-preserving α-renaming.

In the context of the API+LBS architecture, we claim that SES is a practically relevant language for developing both security-critical API code and also for developing third-party applications. Compared to FBJS [82], ADsafe [15] and the ES3 subsets devised in previous sandboxing studies [47, 49] for third-party application development, SES is a more permissive language subset as it includes this, eval and the property access operator $e_1[e_2]$. Furthermore, while SES has a restricted semantics to support isolation, the corresponding restrictions in FBJS are enforced using a combination of filtering, rewriting and wrapping that is not clearly documented in a public standard. In addition, FBJS does not have full lexical scoping or immutable built-in objects. In the future, we believe that the clean language design of SES would be more attractive to third-party application developers than languages such as FBJS that support similar forms of sandboxing via code rewriting and wrapping.

While SES requires programmers of security-critical code to use a more limited

form of ES5, we believe the clean semantic properties of SES and the power of analysis methods enabled by it would provide ample motivation for concerned programmers to adopt this language. We back this claim further in Chapter 8, where we develop an automated tool `ENCAP` for reasoning about confinement properties of SES-light APIs. We show that the ADsafe API implementation can be straightforwardly desugared into SES-light, which in turn suggests that careful programmers may already respect some of the semantically motivated limitations of SES-light.

Chapter 8

API Confinement

Sandboxing mechanisms based on the API+LBS architecture consist of two components: an API that implements a reference monitor to mediate access to security-critical resources and a language-based sandboxing mechanism that ensures that third-party components obtain access to security-critical resources only via the API. While Chapters 5 and 6 focussed on designing provable-correct sandboxing mechanisms, in this chapter we focus on verifying that the API reference monitor correctly mediates access to security-critical resources. This problem is called the *API Confinement* problem. Verifying API confinement requires showing that no sandboxed third-party component can use the API to obtain a direct reference to a security-critical resource. This effectively requires reasoning about *all possible* interleavings of API method calls, which can only be carried out by statically analyzing the API implementation.

In Chapter 7, we stated five key limitations of ES3 that make it unfavorable to static analysis and defensive programming, and proposed a sub-language SES-light of ES5-strict that overcomes these limitations. In this chapter[1] we analyze the language SES-light and develop an automated tool `ENCAP` for statically verifying confinement of APIs written in SES-light. Given an API implementation and a set of security-critical resources, `ENCAP` soundly verifies whether the API confines the resources when subjected to arbitrary third-party SES-light code that only has access to the API —

[1]This chapter is based on joint work with Ulfar Erlingsson, Mark S. Miller and Jasvir Nagra.

essentially third-party SES-light code satisfying *hosting page isolation* (see Chapter 3). While analyzing the API we view such third-party SES-light code as the *attacker*.

We analyzed the November 2010 version of the Yahoo! ADsafe library [15] using `ENCAP`, and found a previously undetected security oversight that could be exploited to leak access to the document object (and hence the entire DOM tree). This demonstrates the value of our analysis, as ADsafe is a mature security filter that has been subjected to several years of scrutiny and even automated analysis [40]. After repairing the vulnerability, our tool is sufficient to prove confinement of the resulting library against the threat model defined in this chapter.

Static analysis method. The main technique used in our verification procedure is a conventional context-insensitive and flow-insensitive points-to analysis. We analyze the API implementation and generate a conservative Datalog model of all API methods. We encode the attacker as a set of Datalog rules and facts, whose consequence set is an abstraction of the set of all possible invocations of all API methods. Our attacker encoding is similar to the encoding of the conventional Dolev-Yao network attacker, used in network protocol analysis. We prove the soundness of our procedure by showing that the Datalog models for the API and the attacker are sound abstractions of the semantics of the API and the set of all possible sandboxed third-party SES-light code satisfying hosting page isolation, respectively. While the specific procedure and the proofs presented in this chapter apply to SES-light, the overall Datalog-based analysis procedure can easily be extended to the complete SES language.

Organization. The rest of this chapter is organized as follows: Section 8.1 formally defines the API confinement problem. Section 8.2 presents our Datalog-based static analysis procedure for verifying confinement of SES-light APIs. Section 8.3 presents applications of the procedure to the Yahoo! ADsafe DOM API and also certain benchmark examples from the object-capability and security literatures, and finally Section 8.4 discusses related work.

8.1 The API Confinement Problem

In this section, we formally define the API confinement problem for SES-light. We begin by defining a sandboxing mechanism for enforcing hosting page isolation on third-party code in SES-light. In defining the problem, we make use of the labelled semantics and the formal analysis framework for SES-light developed in Chapter 7. In what follows, we provide quick recap of the main notations.

A SES-light program state is a triple H, A, t consisting of a heap H, a stack of records A and a statement t. Σ is the universe of all states. The initial SES-light heap and stack are denoted by H_0^{SESl} and A_0^{SESl}. *Vars* is the universe of all user-level SES-light variables and *#global* is the location of the global object. Given states S and T, we say S reaches T in many steps, denoted by $S \rightsquigarrow T$, iff either $S \rightarrow T$ holds or there exists states $S_1, \ldots, S_n$ $(n \geq 1)$ such that $S \rightarrow S_1 \rightarrow \ldots \rightarrow S_n \rightarrow T$ holds. Given a set of states $\mathcal{S}$, $Reach(\mathcal{S})$ is the set of all reachable states, defined as $\{S' \mid \exists S \in \mathcal{S} : S \rightsquigarrow S'\}$.

In the labelled semantics of SES-light, all heap locations, environment records and statements carry labels from the universe $\mathcal{L}$. The labels for heap locations and environment records correspond to the labels of the corresponding terms that created them. To avoid notational overhead, we use the same symbols l, R and s for labelled locations, activation records and statements, and define $Lab(l)$, $Lab(R)$ and $Lab(s)$ respectively as the labels associated with them. The map Lab is naturally extended to sets of heap locations and activation records.

8.1.1 Hosting page Isolation for SES-light

The hosting page isolation property requires that third-party code must only access global variables from a given whitelist $\mathcal{G}$. The whitelist $\mathcal{G}$ is designed so that it contains only those global variables that hold references to API objects. While designing a sandboxing mechanism for enforcing hosting page isolation is challenging for ES3, it is completely straightforward for SES-light, thanks to the cleaner semantics. The global variables accessed by an SES-light term are essentially its free variables. Thus given a third-party SES-light term t the sandboxing mechanism only needs to check

if $Free(t) \not\subseteq \mathcal{G}$. This check can be in fact be carried out by the special *free-variable-restricted* eval statement $\hat{\mathsf{eval}}_{\mathsf{nf}}$ (see Chapter 7). Thus the sandboxing mechanism can be characterized using the following rewriting rule.

Rewrite 4 *For a whitelist $\mathcal{G} = \{m_1, \ldots, m_n\}$, rewrite the third-party term s to* $\hat{\mathsf{eval}}_{\mathsf{nf}}$*("s", m_1, ..., m_n)*

The above rewriting is sufficient for enforcing hosting page isolation on third-party SES-light code.

8.1.2 The Setup

In accordance with the API+LBS architecture, the hosting page code runs first and creates an API object, which is then accessed by sandboxed third-party code that runs next. The hosting page code is called the *trusted API service*. We assume for simplicity that the hosting page stores the API object in some shared global variable api. In order for this mechanism to be secure, third-party code must be appropriately sandboxed so that the only global variable accessible to it is api. In order to set up the confinement problem we also provide third-party code access to a global variable un, which is used as a *test variable* in our analysis and is initially set to undefined. The objective of third-party code is to store a reference to a security-critical resource in it. Thus we sandbox a third-party term so that it can only access global variables named in the whitelist {"api", "un"}. In accordance with the sandboxing mechanism stated in Section 8.1.1, for a third-party term s, the sandboxed term is $\hat{\mathsf{eval}}_{\mathsf{nf}}$("$s$","api","un"). Without loss of generality, we assume that the API service t is suitably-α-renamed according to the procedure in Definition 19, so that it does not use the variable un.

In summary, if t is the trusted API service and s is the third-party term then the overall program that executes in the system is

$$\mathsf{SYS}(t, s, \mathsf{api}, \mathsf{un}) := t;\ \mathsf{var\ un};\ \hat{\mathsf{eval}}_{\mathsf{nf}}(\text{"}s\text{"}, \text{"api"}, \text{"un"})$$

Specifying critical resources. In the setup considered in this chapter, security-critical resources are all of the Document Object Model (DOM) objects, and certain objects used by the API service for holding trusted state. Since DOM objects provide

several properties and methods for manipulating features of the underlying Web page, unrestricted access to them may allow third-party code to arbitrarily alter the page. We therefore conservatively consider all DOM objects as security-critical. The initial root of the DOM tree is pointed to by the "document" property of the global object. Therefore, we conservatively model the entire DOM tree by a single object, held in the "document" property of the global object, with all data properties pointing to itself and all methods pointing to the stub function(x){return document}. In essence, we say that the entire DOM tree leaks via all properties and methods of all DOM objects.

In the labelled semantics of SES-light (see Section 7.3.2), all heap locations are labelled with the label of the code that was responsible for allocating it. Thus, we define the set L_{sec} consisting of labels of all of the DOM objects, and the allocation-site labels of all security-critical objects defined by the API implementation. The goal of the API is to confine all object references that have a label from the set L_{sec}.

8.1.3 Problem Statement

Informally, the API confinement property for a trusted API service t can be stated as: for all statements s, the execution of $\mathsf{SYS}(t, s, \mathsf{api}, \mathsf{un})$ with respect to the initial heap-stack $H_0^{\mathrm{SESl}}, A_0^{\mathrm{SESl}}$ never stores an object with a label from L_{sec} in the variable un. In order to formally define this property, we make use of the *points-to* set of a variable for a given set of program states. The points-to set of a user variable v for a set of states $\mathcal{S}$, denoted by $PtsTo(v, \mathcal{S})$, is the set of labels associated with the values of property "v" of the global object for each state in $\mathcal{S}$. Recall that $Lab(l)$ provides the label associated with a location l.

Definition 20 *[Points-to] Given a set of states $\mathcal{S} \in 2^{\Sigma}$, and a variable $v \in Vars$, $PtsTo(v, \mathcal{S})$ is the set: $\{Lab(H(\#global).\text{"}v\text{"}) \mid \exists H, A, t : H, A, t \in \mathcal{S}\}$.*

Given a trusted API service t, the set of all possible initial states is given by:

$$\mathcal{S}_0(t) := \{H_0, A_0, \mathsf{SYS}(t, s, \mathsf{api}, \mathsf{un}) \mid s \in Stmts_{user}^{\mathrm{SESl}}\}$$

The API Confinement property is then formally defined as follows.

Definition 21 *[Confinement Property] A trusted API service t safely encapsulates a set of security-critical object labels L_{sec} iff $PtsTo(\text{"un"}, Reach(\mathcal{S}_0(t))) \cap L_{sec} = \emptyset$. This property is denoted by $Confine(t, L_{sec})$.*

We now formally state the API Confinement problem

> *Given a trusted API service t and a set of securit-critical object labels L_{sec}, verify $Confine(t, L_{sec})$.*

8.2 Analysis Procedure

In this section we define a procedure $\mathcal{D}(t, L_{sec})$ for verifying that an API service t safely confines a set of critical resources L_{sec}. The main idea is to define a tractable procedure for over-approximating the set $PtsTo(\text{"un"}, Reach(\mathcal{S}_0(t)))$, which is the set of values pointed to by by the variable "un" in the set of all states reachable from the initial states $\mathcal{S}_0(t)$. We adopt an *inclusion-based*, *flow-insensitive* and *context-insensitive* points-to analysis technique [4] for over-approximating this set. This is a well-studied and scalable points-to analysis technique. Flow-insensitivity means that the analysis is independent of the ordering of the statements and context-insensitivity means that the analysis only models a single activation record that is shared across all function calls. Given the presence of closures and absence of a static call graph in SES-light, a context-sensitive analysis is known to be significantly more expensive than a context-insensitive one (see [31, 56] for complexity results). In this dissertation we therefore adopt a context-insensitive analysis which is polynomial time. Given that there has been very little prior work (see [28]) on defining provable-sound static analyses for subset of JavaScript at the scale of SES-light, we believe that a provably-sound flow-insensitive and context-insensitive analysis is a reasonable first step.

In adopting the well-known *inclusion-based* based flow-insensitive and context-insensitive points-to analysis technique to our problem, we are faced with the following challenges: (1) Statically encoding $\hat{\text{eval}}_{\text{nf}}$ statements, (2) Statically reasoning about the entire set of states $\mathcal{S}_0(t)$ at once, and (3) Soundly modeling the various non-standard features of SES-light. We resolve these challenges as follows. Recall that

the arguments to $\hat{\mathsf{eval}}_{\mathsf{nf}}$ statically specify a bound on the set of free variables of the code being eval-ed. We use this bound to define a worst case encoding for $\hat{\mathsf{eval}}_{\mathsf{nf}}$ calls, which essentially amounts to creating all possible points-to relationships between all the objects reachable from the set of free variables. Since the encoding only depends on the set of free variables and is independent of the actual code being evaluated, it resolves both challenges (1) and (2). For challenge (3), we leverage upon the insights gained while developing the formal semantics for SES-light (see Chapter 7) and formulate our abstractions in a sound manner. We also back our procedure with a proof of correctness which guarantees that we (conservatively) respect the semantics. We follow the approach of Whaley et al. [91] and express our analysis algorithm in Datalog. Before describing the details of our procedure, we provide a quick introduction to Datalog.

Quick introduction to Datalog. A Datalog program consists of *facts* and *inference rules*. Facts are of the form $P(t_1, \ldots, t_n)$ where P is a predicate symbol and t_i are terms, which could be constants or variables. Rules are sentences that provide a means for deducing facts from other facts. Rules are expressed as *horn clauses* with the general form $L_0 :- L_1, \ldots, L_n$ where $L_0, \ldots, L_n$ are facts. Given a set of facts $\mathcal{F}$ and a set of inference rules $\mathcal{R}$, $Cons(\mathcal{F}, \mathcal{R})$ is the set of all "consequence" facts that can be obtained by successively applying the rules to the facts, upto a fixed point. As an example if $\mathcal{F} := \{edge(1,2), edge(2,3)\}$ and

$$\mathcal{R} := \left\{ \begin{array}{l} path(x,y) \ :- \ edge(x,y); \\ path(x,z) \ :- \ edge(x,y), path(y,z) \end{array} \right\}$$

then $Cons(\mathcal{F}, \mathcal{R})$ is the set $\{edge(1,2), edge(2,3), \ path(1,2), path(2,3), path(1,3)\}$. We refer the reader to [12] for a comprehensive survey of Datalog and its semantics.

Procedure Overview. A high-level overview of the procedure $\mathcal{D}(t, L_{sec})$ is as follows:

(1) Pick *any* $s \in Stmts^{\text{SESl}}_{user}$, encode the statement $\mathsf{SYS}(t, s, \mathsf{api}, \mathsf{un})$ as a set of Datalog facts and add them to a Database

(2) Conservatively encode the semantics of SES-light in the form of Datalog inference rules.

(3) Compute the consequence set of the Database from (1), using the inference rules from (2), to obtain a Database of all consequence facts.

(4) Analyze the Database from (3) for any confinement violating facts.

The rest of this section is organized as follows: 8.2.1 describes the encoding of SES-light statements as Datalog facts, 8.2.2 presents the inference rules, 8.2.3 presents the formal definition of the procedure and 8.2.4 provides a soundness argument.

8.2.1 Datalog Relations and Encoding

Our encoding of program statements into Datalog facts, makes use of the standard abstraction of heap locations as allocation-site labels. Since JavaScript represents objects and function closures in the same way, this applies to function closures as well. In the terminology of control-flow analysis, this abstraction makes our analysis 0-CFA as all closures allocated at the same allocation-site in code are abstracted by the same abstract element (which is the label for that allocation-site). Furthermore, the analysis only supports *weak updates*, which means we aggregate values with each variable and property assignment.

Facts are expressed over a fixed set of relations $\mathbb{R}$, enumerated in Figure 8.1 along with their domains. V is the domain for variable and property names, L is the domain for allocation-site labels (abstract locations) and I is the domain for function argument indices. We assume that $V \subseteq \mathit{Vars}$ and $L \subseteq \mathcal{L}$. A similar set of relations has been used for points-to analysis of Java in [91, 8]. Besides relations that capture facts about the program, we use *Heap*, *Stack*, *Prototype* to capture facts about the heap and stack. Fact $\mathit{Heap}(\hat{l}_1, x, \hat{l}_2)$ encodes that an object with label $\hat{l}_1$ has a field x pointing to an object with label $\hat{l}_2$, fact $\mathit{Stack}(x, \hat{l})$ encodes that variable x points to an object with label $\hat{l}$, and $\mathit{Prototype}(\hat{l}_1, \hat{l}_2)$ encode stat object with label $\hat{l}_1$ has a prototype with label $\hat{l}_2$. We define *Facts* as the set of all possible facts that can be expressed over the relations in $\mathbb{R}$.

Relations for encoding programs:

$Assign : 2^{V \times V}$	$Throw : 2^{L \times V}$
$Load : 2^{V \times V \times V}$	$Catch : 2^{L \times V}$
$Store : 2^{V \times V \times V}$	$Global : 2^{V}$
$FormalArg : 2^{L \times I \times V}$	$Annotation : 2^{V \times V}$
$FormalRet : 2^{L \times V}$	$ObjType : 2^{L}$
$Instance : 2^{L \times V}$	$FuncType : 2^{L}$
$ArrayType : 2^{L}$	$NotBuiltin : 2^{L}$
$Actual : 2^{V \times I \times V \times V \times L}$	

Relations for encoding the heap-stack:

$Heap : 2^{L \times V \times L}$	$Stack : 2^{V \times L}$
$Prototype : 2^{L \times L}$	

Figure 8.1: Encoding relations for SES-light

We now describe the encoding of user statements into facts over $\mathbb{R}$. Since we have the same domain V for variable and property names, while defining the encoding we convert all property name strings m to the corresponding identifier $sToi(m)$ We use $AnnFacts(m) := \{Annotation(sToi(m), sToi(an)) \mid an \in \mathcal{A} \wedge Ann(m) = an\}$ as the set of all annotation facts for the string m. For each label $\hat{l}$, we assume a unique and countably-infinite set of labels $h(\hat{l}, 1),\ h(\hat{l}, 2),\ \ldots$ associated with it. The purpose of these labels is to denote objects that get created "on the fly" during the execution of a statement. We also assume a countable-infinite set of temporary variables $\$, \$_1, \ldots$.

The encoding of a statement s depends on the label $\hat{l}$ of the nearest enclosing scope in which it appears and is expressed by the map $Enc_t(s, \hat{l})$, defined formally in Figures 8.2, 8.3, 8.4 and 8.5. In the rest of this section, we comment on the definition of Enc_t for some key statements. The definition is based on the labeled semantics of SES-light (see Section 7.3.2).

Assign. For assignment statements $y = x$, we simply record that the contents of variable x flows into variable y, using the fact $Assign(y, x)$.

Binary Expression Statement. According to the semantics of a binary operation statement $s := y = x_1$ BIN x_2, if BIN $\in \{\&\&, ||\}$ and if x_1 or x_2 resolve to an object

then they could potentially get assigned to y. We therefore conservatively encode such statements by $\{Assign(y, x_1), Assign(y, x_2)\}$. On the other hand, if $BIN \notin \{\&\&, ||\}$ and if x_1 or x_2 resolve to an object, then the evaluation might trigger an implicit type conversion of these objects to primitive values. We therefore encode such statements by $\{TP(x_1, \hat{l}), TP(x_2, \hat{l})\}$, where $TP(x, \hat{l})$ implies that a *to-primitive* conversion must be triggered on objects stored in variable x in a scope labeled $\hat{l}$ (modeled by inference rules [TP1] and [TP2]). We found that these subtle semantic features of binary expression statements are not captured by existing JavaScript points-to analysis frameworks [25, 35].

Load. The evaluation of a load statement $s := y = x_1[x_2, an]$ could potentially trigger a to-primitive conversion on the object stored in x_2 and a to-object conversion on the value stored in x_1. This is encoded by the following set of facts

$$\{TP(x_2, \hat{l}), Stack(x_1, \hat{l}_1), ObjType(\hat{l}_1), NotBuiltin(\hat{l}_1)\} \cup \{Load(y, x_1, sToi(an))\}$$

where $\hat{l}_1 = h(Lab(s), 1)$ is the abstract location of the object created from the to-object conversion of value stored in variable x_1. The first set in the union encodes that x_1 points to a non-built-in object with abstract location $\hat{l}_1$ and that x_2 must be converted to a primitive value in the scope $\hat{l}$. The second set encodes that the value of the expression $x_1.p$ flows into y for all property names p that annotate to an.

Function Declaration. A function declaration $s :=$ function $x(\tilde{y_i})\{s\}$ is encoded as:

$$\left\{ \begin{array}{l} FormalArg(\hat{l}_1, 1, y_1), \ldots, FormalArg(\hat{l}_1, n, y_n), \\ FormalArg(\hat{l}_1, \text{“}a\text{”}, \mathsf{arguments}), FuncType(\hat{l}_1), \\ FormalArg(l_1, \text{“}t\text{”}, \mathsf{this}), Stack(x, \hat{l}_1) \\ ObjType(\hat{l}_2), Heap(\hat{l}_1, \mathsf{prototype}, \hat{l}_2) \end{array} \right\} \bigcup Enc_t(s_1, \hat{l}_1)$$

where $\hat{l}_1 = h((Lab(s), 1)$ and $\hat{l}_2 = h((Lab(s), 2)$ are abstract locations for the function and prototype objects that get created dynamically. *FormalArg* encodes the positions of all the formal arguments, including default arguments this and arguments whose positions are denoted by “t” and “a” respectively[2].

Function Call. A function call statement $s :=$ $y = x(\tilde{x_i})$ is similarly encoded using

[2]We assume that “t”, “a” $\in I$.

facts of the form $Actual(x, i, x_i, y, \hat{l})$ where x is the variable holding the function object, x_i is the actual argument at position i, y is the return variable, and $\hat{l}$ is the label of the nearest enclosing scope. The actual arguments object is an array containing the values stored in variables $\tilde{x_i}$. We use a new abstract label $\hat{l}_1 = h((Lab(s), 1)$ for the location of the arguments object, a variable \$ to store it, and the index "a" for the argument position on the call site. We use the annotation $\$Num$ for all properties of the arguments array. Finally, the encoding for the call statement is given by:

$$\left\{ \begin{array}{l} ArrayType(\hat{l}_1), Stack(\$, \hat{l}_1), Actual(x, \text{“}a\text{”}, \$, y, \hat{l}), \\ Store(\$, \$Num, x_1), \ldots, Store(\$, \$Num, x_n) \\ Actual(x, 1, x_1, y, \hat{l}), \ldots, Actual(x, n, x_n, y, \hat{l}) \end{array} \right\}$$

Free-variable-restricted eval. As discussed in Chapter 7, the evaluation of the free-variable-restricted eval statement $s := \hat{\mathsf{eval}}_{\mathsf{nf}}(\mathsf{x}, \tilde{m})$ forces the free variable names of the code being eval-ed to be contained in $\{\tilde{sToi}(m)\}$. Since we do not know the code statically, we conservatively assume that all possible points-to relationships are created between all objects reachable from the free and bound variables. To make the encoding finite, we summarize all the bound variables by a single variable $\alpha(\mathsf{x}_{\mathsf{eval}}, Lab(s))$ where $\mathsf{x}_{\mathsf{eval}}$ is a constant variable associated with $\hat{\mathsf{eval}}_{\mathsf{nf}}$ statements, and α is a variable-renaming map (see Section 7.3.4 in Chapter 7). We summarize all locally allocated objects by a single abstract location $\hat{l}_1 = h(Lab(s), 1)$. For the enclosing scope $\hat{l}$, the encoding is given by the set $Eval(\hat{l}, \hat{l}_1, \{\alpha(\mathsf{x}_{\mathsf{eval}}, Lab(s)), \tilde{sToi}(m)\})$, defined formally in Figure 8.5. The set is obtained by instantiating all relations with all possible valid combinations of the variables in the set $V_e = \{\alpha(\mathsf{x}_{\mathsf{eval}}, Lab(s)), \tilde{sToi}(m)\}$ and locations in the set $L_e = \{\hat{l}, \hat{l}_1\}$.

Built-in Objects and DOM. We encode all built-in objects and DOM objects present on the initial heap H_0 as a set of facts and rules $\mathcal{I}_0$. For all references l_1, l_2, and properties x such that $H_0(l_1).x = l_2$, $\mathcal{I}_0$ contains the fact $Heap(Lab(l_1), x, Lab(l_2))$. For all properties v and locations l, such that property p of the global object $\#global$ points to location l, $\mathcal{I}_0$ contains the facts $Global(v)$ and $Stack(v, Lab(l))$. For each built-in method, $\mathcal{I}_0$ contains appropriate inference rules over $Actual$ facts that capture the semantics of the method. We give the rules for the Function.prototype.apply method,

labeled by $\hat{l}_{apply}$, as an example. According to the semantics of the apply method, the call x_0.apply(x_1,x_2) involves calling the function pointed to by x_0 with this value x_1 and arguments as stored on the array x_2. It is encoded by the following inference rules

$$\begin{aligned} Actual(x_0, \text{“}t\text{”}, x_1, y, \hat{l}_{apply}) \ :- \ & Actual(x, \text{“}t\text{”}, x_0, y, \hat{l}_1), Actual(x, 1, x_1, y, \hat{l}_1), \\ & Stack(x, \hat{l}_{apply}) \\ Actual(x_0, i, x_3, y, \hat{l}_{apply}) \ :- \ & Actual(x, \text{“}t\text{”}, x_0, y, \hat{l}_1), Actual(x, 2, x_2, y, \hat{l}_1), \\ & Heap(x_2, \$Num, x_3), Stack(x, \hat{l}_{apply}) \end{aligned}$$

Encoding built-in methods using rules provides much better call-return matching than the naive encoding using *FormalArg* facts. This turned out to be very useful in our experiments as calls to built-in methods are pervasive in most API definitions. For all built-in prototype objects, $\mathcal{I}_0$ also contains rules for capturing the inheritance relation. For example, the following rule is used for the Object.prototype object which is labelled as $\hat{l}_{oProt}$.

$$Prototype(\hat{l}, \hat{l}_{oProt}) :- ObjType(\hat{l})$$

DOM methods are encoded by encoding the function declaration function($\tilde{x}$){return document}.

8.2.2 Inference Rules

We now briefly describe the set of inference rules $\mathcal{R}$, which model a flow-insensitive and context-insensitive semantics of SES-light. The rules are formally defined in Figure 8.6. Since it is clear from the context, we elide the hat and use symbols l, m, n, k for labels. We use symbols x, y, z, f, a for variables and property names, and symbol i for fields names.

Assign, Load and Store. Rules [Assign], [Load] and [Store1] are straightforward and model the semantics of assignments, load and store statements. Rules [Prototype1] and [Prototype2] conservatively flatten all prototype chains by taking the reflexive and transitive closure of the relation *Prototype*. Rules [Store2] and [Store3] capture that an annotated property store gets propagated on all the concrete property names that satisfy the annotation.

$Enc_t : Terms^{\text{SESl}}_{user} \times \mathcal{L} \to 2^{Facts}$

Given a user-statement s and a label $\hat{l}$, $Enc_t(s, \hat{l})$ is recursively defined by the following table.

Let $\hat{l}_i = h(Lab(s), i)$ and $\$_1, \ldots$ are free variables from V.

$\mathbf{s}$	$Enc_{\mathbf{t}}(\mathbf{s}, \hat{\mathbf{l}})$
$y = x$	$\{Assign(y, x)\ \}$
$y = x_1\ BIN\ x_2$	$\{Assign(y, x_1), Assign(y, x_2)\}$ iff $BIN \in \{\&\&, \|\|\}$
$y = x_1\ BIN\ x_2$	$\{TP(x_1, \hat{l}), TP(x_2, \hat{l})\}$ iff $\mathsf{BIN} \notin \{\&\&, \|\|\}$
$y = UN\ x_1$	$\{TP(x_1, \hat{l})\}$
$y = \mathsf{delete}\ x_1[x_2, an]$	$\{TP(x_2, \hat{l}), Stack(x_1, \hat{l}_1), NotBuiltin(\hat{l}_1)\}$
$y = \mathsf{delete}\ x_1[m, an]$	$\{Stack(x_1, \hat{l}_1), NotBuiltin(\hat{l}_1)\}$
$y = x_1[x_2, an]$	$\{TP(x_2, \hat{l}), Stack(x_1, \hat{l}_1), NotBuiltin(\hat{l}_1), Load(y, x_1, sToi(an))\}$
$y = x_1[m, an]$	$\{Stack(x_1, \hat{l}_1), NotBuiltin(\hat{l}_1), Load(y, x_1, sToi(m))\}\ \cup\ AnnFacts(m)$
$x_1[x_2, an] = x_3$	$\left\{\ TP(x_2, \hat{l}), TP(x_3, \hat{l}), Store(x_1, sToi(an), x_3)\ \right\}$
$x_1[m, an] = x_3$	$\{TP(x_2, \hat{l}), TP(x_3, \hat{l}), Store(x_1, sToi(m), x_3)\}\ \cup\ AnnFacts(m)$
$y = \{m_i \tilde{:} x_i\}$	$\left\{ \begin{array}{l} ObjType(\hat{l}_1), Stack(y, \hat{l}_1), \\ Store(y, sToi(m_1), x_1), \ldots, Store(y, sToi(m_n), x_n) \end{array} \right\}$ $\bigcup\ AnnFacts(m_1)\ \cup\ \ldots\ \cup\ AnnFacts(m_n)$
$y = [\tilde{x_i}]$	$\{ArrayType(\hat{l}_1), Stack(y, \hat{l}_1), Store(y, \$\mathsf{Num}, x_1), \ldots, Store(y, \$\mathsf{Num}, x_n)\}$
$y = x[x', an](\tilde{x_i})$	$\left\{ \begin{array}{l} Load(\$_1, x, sToi(an)), ArrayType(\hat{l}_1), Stack(\$_2, \hat{l}_1), \\ Store(\$_2, \$\mathsf{Num}, x_1), \ldots, Store(\$_2, \$\mathsf{Num}, x_n), \\ Actual(\$_1, \text{“}a\text{”}, \$_2, y, \hat{l}), Actual(\$_1, \text{“}t\text{”}, x, y, \hat{l}), \\ Actual(x, 1, x_1, y, \hat{l}), \ldots, Actual(x, n, x_n, y, \hat{l}) \end{array} \right\}$

Figure 8.2: Encoding of SES-light statements (Part 1)

$\mathbf{s}$	$Enc_{\mathbf{t}}(\mathbf{s}, \hat{\mathbf{l}})$
$y = x[m,an](\tilde{x_i})$	$\left\{ \begin{array}{l} Load(\$_1, x, sToi(m)), ArrayType(\hat{l}_1), Stack(\$_2, \hat{l}_1), \\ Store(\$_2, \$\mathsf{Num}, x_1), \ldots, Store(\$_2, \$\mathsf{Num}, x_n), \\ Actual(\$_1, \text{“}a\text{”}, \$_2, y, \hat{l}), Actual(\$_1, \text{“}t\text{”}, x, y, \hat{l}) \\ Actual(x, 1, x_1, y, \hat{l}), \ldots, Actual(x, n, x_n, y, \hat{l}) \end{array} \right\}$ $\bigcup\ AnnFacts(m)$
$y = x(\tilde{x_i})$	$\left\{ \begin{array}{l} ArrayType(\hat{l}_1), Stack(\$, \hat{l}_1), Actual(x, \text{“}a\text{”}, \$, y, \hat{l}), \\ Store(\$, \$\mathsf{Num}, x_1), \ldots, Store(\$, \$\mathsf{Num}, x_n) \\ Actual(x, 1, x_1, y, \hat{l}), \ldots, Actual(x, n, x_n, y, \hat{l}) \end{array} \right\}$
$y = \mathsf{new}\ x(\tilde{x_i})$	$\left\{ \begin{array}{l} ArrayType(\hat{l}_1), Stack(\$, \hat{l}_1), ObjType(\hat{l}_2), \\ Store(\$, \$\mathsf{Num}, x_1), \ldots, Store(\$, \$\mathsf{Num}, x_n) \\ Stack(y, \hat{l}_2), Actual(x, \text{“}a\text{”}, \$, y, \hat{l}), Instance(\hat{l}_2, x), \\ Actual(x, i, x_i, y, \hat{l}), Actual(x, \text{“}t\text{”}, y, y, \hat{l}) \end{array} \right\}$
$y = \mathsf{function}\ x(\tilde{y_i})\{s_1\}$	$\left\{ \begin{array}{l} FormalArg(\hat{l}_1, 1, y_1), \ldots, FormalArg(\hat{l}_1, n, y_n), \\ FormalArg(\hat{l}_1, \text{“}a\text{”}, \mathsf{arguments}), FuncType(\hat{l}_1), \\ FormalArg(l_1, \text{“}t\text{”}, \mathsf{this}), Stack(x, \hat{l}_1), Stack(y, \hat{l}_1) \\ ObjType(\hat{l}_2), Heap(\hat{l}_1, \mathsf{prototype}, \hat{l}_2) \end{array} \right\}$ $\bigcup\ Enc_t(s_1, \hat{l}_1)$
$\mathsf{function}\ x(\tilde{y_i})\{s\}$	$\left\{ \begin{array}{l} FormalArg(\hat{l}_1, 1, y_1), \ldots, FormalArg(\hat{l}_1, n, y_n), \\ FormalArg(\hat{l}_1, \text{“}a\text{”}, \mathsf{arguments}), FuncType(\hat{l}_1), \\ FormalArg(l_1, \text{“}t\text{”}, \mathsf{this}), Stack(x, \hat{l}_1) \\ ObjType(\hat{l}_2), Heap(\hat{l}_1, \mathsf{prototype}, \hat{l}_2) \end{array} \right\}$ $\bigcup\ Enc_t(s_1, \hat{l}_1)$

Figure 8.3: Encoding of SES-light statements (Part 2)

$\mathbf{s}$	$Enc_{\mathbf{t}}(\mathbf{s}, \hat{\mathbf{l}})$
$\hat{\mathsf{eval}}_{\mathsf{nf}}(x,\tilde{m})$	$Eval(\hat{l}, Lab(s), \{\alpha(\mathsf{x}_{\mathsf{eval}}, Lab(s)), \tilde{sToi}(m)\})$
return x	$\{FormalRet(\hat{l}, x)\}$
var x	$\emptyset$
throw x	$\{Throw(\hat{l}, x)\}$
s; t	$Enc_t(s, \hat{l}) \cup Enc_t(t, \hat{l})$
if (e) then s else t	$Enc_t(s, \hat{l}) \cup Enc_t(t, \hat{l})$
while (e) s	$Enc_t(s, \hat{l})$
for $(x_1$ in $x_2)$ s	$\{PrimType(Lab(s)), Stack(x_2, Lab(s)\} \cup Enc_t(s, \hat{l})$
try$\{s_1\}$catch$(x)\{s_2\}$finally$\{s_3\}$	$\{Catch(\hat{l}, x)\} \cup Enc_t(s_1, \hat{l}) \cup Enc_t(s_2, \hat{l}) \cup Enc_t(s_3, \hat{l})$

Figure 8.4: Encoding of SES-light statements (Part 3)

$Eval : \mathcal{L} \times \mathcal{L} \times 2^{Vars} \rightarrow 2^{Facts}$

Given labels l_{outer}, l_{obj} and a set of user-variables V_e, $Eval(l_{outer}, l_{obj}, V_e\})$ is defined as:

$$\{Assign(x_1, x_2) \mid x_1, x_2 \in V_e\} \cup \{Load(x_1, x_2, \$\mathsf{All}) \mid x_1, x_2 \in V_e\}$$

$$\cup \{Store(x_1, \$\mathsf{All}, x_2) \mid x_1, x_2 \in V_e\}$$

$$\cup \{Actual(x_1, i, x_2, x_3, l) \mid x_1, x_2, x_3 \in V_e; l \in L_e\} \cup \{FormalArg(l_{obj}, i, v) \mid v \in V_e\}$$

$$\cup \{FormalRet(l_{obj}, v) \mid v \in V_e\} \cup \{Instance(l_{obj}, v) \mid v \in V_e\}$$

$$\cup \{Throw(l, v) \mid v \in V_e; l \in L_e\} \cup \{Catch(l, v) \mid v \in V_e; l \in L_e\}$$

$$\cup \{NotBuiltin(l_{obj})\} \cup \{FuncType(l_{obj})\} \cup \{ArrayType(l_{obj})\} \cup \{ObjType(l_{obj})\}$$

where $L_e := \{l_{obj}, l_{outer}\}$

Figure 8.5: Encoding for the $\hat{\mathsf{eval}}_{\mathsf{nf}}$ statement

To-Primitive. Rules [TP1] and [TP2] model the semantics of to-primitive type conversions. Given a fact $TP(x, l)$, the rule over-approximates the behavior of a to-primitive type conversion by deriving a call to the "toString" and "valueOf" methods of all objects stored at x. Since the value returned by a to-primitive conversion is primitive, it is discarded by specifying a temporary variable $\$_{\mathsf{d}}$ as the return variable.

Function Calls. Function calls are handled by rules [ACTUAL1], [ACTUAL2] and [ACTUAL3]. Since functions are modelled as objects in JavaScript, call targets are also resolved via the heap and stack. The rule [ACTUAL1] flows actual parameters to formal parameters, [ACTUAL2] flows formal return values to actual return values and [ACTUAL3] propagates "throws" across the call chain.

Global and Catch Variables. Since global variables are properties of the global object, assignments to global variables are propagated as property stores on the global object and vice versa. This is modeled by rules [GLOBAL1] and [GLOBAL2]. The rule [CATCHVAR] conservatively flows "throws" from a particular scope into all "catch" variables appearing in that scope.

8.2.3 Procedure for Verifying API Confinement

The procedure $\mathcal{D}(t, L_{sec})$ for verifying that an API service t confines a set of security-critical object labels L_{sec} is defined in Figure 8.7. It uses the global object label $\hat{l}_g$ and the abstract points-to map $PtsTo_{\mathcal{D}} : Vars \times 2^{Facts} \rightarrow 2^{\mathcal{L}}$ defined below.

Definition 22 *[Abstract Points-to] Give a set of facts $\mathcal{F} \in 2^{Facts}$ and a variable $v \in Vars$, $PtsTo_{\mathcal{D}}(v, \mathcal{F})$ is defined as $\{\hat{l} \mid Stack(v, \hat{l}) \in \mathcal{F}\}$.*

The first step of the procedure is to pick any program s and encode the term $\mathsf{SYS}(t, s, \mathsf{api}, \mathsf{un})$ in global scope. Given the way $\widehat{\mathsf{eval}}_{\mathsf{nf}}$ statements are encoded, the encoding of the above term does not depend on the term s. The next step is to compute the set of all possible consequences of the encoded facts, under the inference rules $\mathcal{R}$ defined in Figure 8.6. The final step is to compute the abstract points-to set of the variable un from this consequence set and check if it contains any labels from the set L_{sec}. Since the maps Enc_t, $Cons$ and $PtsTo_{\mathcal{D}}$ are computable, the procedure

$Stack(x, l)$	:–	$Stack(y, l), Assign(x, y)$	[ASSIGN]
$Stack(x, n)$	:–	$Load(x, y, f), Prototype(l, m),$ $Heap(m, f, n), Stack(y, l)$	[LOAD]
$Heap(l, f, m)$	:–	$Store(x, f, y), Stack(x, l), NotBuiltin(l),$ $Stack(y, m)$	[STORE1]
$Store(x, a, y)$	:–	$Store(x, f, y), Annotation(f, a)$	[STORE2]
$Store(x, f, y)$	:–	$Store(x, a, y), Annotation(f, a)$	[STORE3]
$Annotation(f, \$\mathsf{All})$			[ANNOTATION]
$Actual(n, \text{“}t\text{”}, x, \$_{\mathsf{d}}, k)$	:–	$TP(x, k), Stack(x, l), Prototype(l, m),$ $Heap(m, \mathsf{toString}, n), FuncType(n)$	[TP1]
$Actual(n, \text{“}t\text{”}, x, \$_{\mathsf{d}}, k)$	:–	$TP(x, k), Stack(x, l), Prototype(l, m),$ $Heap(m, \mathsf{valueOf}, n), FuncType(n)$	[TP2]
$Assign(y, z)$	:–	$Actual(f, i, z, x, k), Stack(f, l),$ $FormalArg(l, i, y)$	[ACTUAL1]
$Assign(x, y)$	:–	$Actual(f, i, z, x, k), Stack(f, l),$ $FormalRet(l, y)$	[ACTUAL2]
$Throw(k, x)$	:–	$Actual(f, i, y, z, k), Stack(f, l), Throw(l, x)$	[ACTUAL3]
$Prototype(l, l)$			[PROTOTYPE1]
$Prototype(l, n)$	:–	$Prototype(l, m), Prototype(m, n)$	[PROTOTYPE2]
$Prototype(l, q)$	:–	$Instance(l, y), Stack(y, m),$ $Prototype(m, n), Heap(n, \mathsf{prototype}, q)$	[PROTOTYPE3]
$Heap(\hat{l}_g, f, l)$	:–	$Stack(f, l), Global(f)$	[GLOBAL1]
$Stack(f, l)$	:–	$Heap(\hat{l}_g, f, l), Global(f)$	[GLOBAL2]
$Assign(x, y)$	:–	$Catch(k, x), Throw(k, y)$	[THROW]

Figure 8.6: Inference rules ($\mathcal{R}$) for SES-light

Procedure $\mathcal{D}(t, L_{sec})$:

(1) *Pick any* $s \in Stmts_{user}^{SESl}$ *and compute* $\mathcal{F}_0(t) = Enc_t(\mathsf{SYS}(t, s, \mathit{api}, \mathit{un}), \hat{l}_g) \cup \mathcal{I}_0$.

(2) *Compute* $\mathcal{F} = Cons(\mathcal{F}_0(t), \mathcal{R})$.

(3) *Check that* $PtsTo_{\mathcal{D}}(\text{“un”}, \mathcal{F}) \cap L_{sec} = \emptyset$.

Figure 8.7: Procedure for verifying $Confine(t, L_{sec})$

is decidable. The procedure is listed purely from the correctness standpoint and does make any efficiency considerations.

8.2.4 Soundness

We prove soundness of the procedure $\mathcal{D}(t, L_{sec})$ by showing that for all statements t and security-critical object labels L_{sec}, $\mathcal{D}(t, L_{sec}) \implies Confine(t, L_{sec})$.

Theorem 10 *[Soundness] For all statements* t *and security-critical object labels* L_{sec}, $\mathcal{D}(t, L_{sec}) \implies Confine(t, L_{sec})$.

The proof of the above theorem is very similar to the one given by Midtgaard et al. in [55] for soundness of 0-CFA analysis. The crux of the proof is in defining a map $Enc : 2^{\Sigma} \to 2^{Facts}$ (abstraction map) for encoding a set of program states as a set of Datalog facts, and showing that for any set of states, the set of consequence facts safely over-approximates the set of reachable states, under the encoding. Due to the sheer volume of the semantics, we only describe a sketch of the proof in the appendix (Section A.5).

8.3 Applications

In this section, we demonstrate the value of our analysis procedure by analyzing three benchmark examples: Yahoo! ADsafe library [15], the Sealer-Unsealer mechanism ([38, 69]) and the Mint mechanism [59]. These are all APIs that have been designed

with an emphasis on robustness and simplicity, and have been previously subjected to security analysis. We analyze these APIs under the SES-light semantics and threat model. The goal of our experiments was to test the effectiveness of the procedure $\mathcal{D}(t, L_{sec})$ defined in Figure 8.7 by checking if it could correctly prove confinement properties for these well-studied APIs.

Analyzer Architecture. We implemented the procedure $\mathcal{D}(t, L_{sec})$ from Figure 8.7 in the form of a tool named `ENCAP`. The tool has a SES-light parser at the front end and the `bddbddb` Datalog engine [90] at the back end. Given an input API definition and a set of security-critical object labels, the parser generates an SES-light AST which is then encoded into a set of Datalog facts. As described in the procedure, this encoding is combined with the encoding of the initial heap and the encoding of the statement $\mathsf{SYS}(t, s, \text{“api”}, \text{“un”})$ for any statement $s \in Stmts_{user}^{\text{SESl}}$.

8.3.1 ADsafe

Our first application is the November 2010 version of Yahoo! ADsafe, which we denote by ADSafe_{Nov10}. As described in Chapter 3, ADsafe follows the API+LBS architecture, with the API being the ADsafe DOM API and the sandboxing mechanism being the JSLint static analyzer. One of the goals of JSLint is to enforce hosting page isolation, that is, to ensure that JavaScript code that passes through it only accesses DOM objects via the ADsafe DOM API. In Section 8.1.1, we have shown that hosting page isolation can be achieved for SES-light by simply rewriting every untrusted third-party program s to $\hat{\mathsf{eval}}_{\mathsf{nf}}(s, \text{“api”})$, where “api” stores a reference to the ADSafe_{Nov10} API object. In our experiments, we therefore focus on analyzing confinement for the ADSafe_{Nov10} API against third-party code restricted using the SES-light sandboxing mechanism. We call this the *SES-light threat model*.

Desugaring the APi and adding annotations. Although the ADSafe_{Nov10} API was implemented in ES3, we found that it did not use setters/getters and eval. As a result we were able to de-sugar it into semantically equivalent SES-light code and thereby make it amenable to confinement analysis using `ENCAP`. In order to make our analysis precise and to support certain JSLint restrictions on untrusted code, we

add suitable property annotations to the API implementation and to the encoding of $\hat{\text{eval}}_{\text{nf}}$ statements. The API reserves certain property names to hide security-critical objects and other book-keeping information. These property names are blacklisted and JSLint filters out all untrusted programs that access any blacklisted property. We support this restriction in our analysis by annotating all *Load* and *Store* facts in the encoding of $\hat{\text{eval}}_{\text{nf}}$ statements with the annotation $Safe which ensures that the property name is not blacklisted. The annotation $Safe is also added to patterns of the form if (!reject(name)){ ... object[name] ... } in the library implementation, where reject is a function in the ADSafe$_{Nov09}$ API implementation that checks if name is blacklisted. The other annotation used is $Num which is added to property lookups involving loop index variables.

Attack. We ran ENCAP on the ADsafe library (approximately 1700 loc) and found that it leaks the document object via methods named "lib" and "go". The running time of the analysis was 5 minutes and 27 seconds, on a standard Linux workstation with 8GB RAM. After analyzing the methods, we were able to construct an actual client program that used them to directly access the document object, thus confirming the leak to be a true positive. The exploit code is present in Figure 8.8.

In order to explain the root cause of the attack, we describe the methods "go" and "lib". The method "go" takes a string id and a function f as arguments. It invokes the function f with objects dom, and adsafe_lib. The dom object has methods that wrap the original DOM methods, and the adsafe_lib object stores libraries defined by untrusted code. The adsafe_lib object is populated with a method "lib" defined as function (name, f) {adsafe_lib[name] = f(adsafe_lib);}. One of the confinement mechanisms used by the API is to virtualize the DOM by creating fake DOM objects that hide the original DOM objects behind the "___node___" property which is then blacklisted by JSLint. This mechanism is broken by the lib method which allows third-party code to write to the "___node___" property of the adsafe_lib object. This is the heart of the exploit. Malicious third-party code installs its own (malicious) function in the "___node___" property of the adsafe_lib object, and then hands the adsafe_lib object to a DOM wrapper method (value in this case) as a fake DOM object, thereby obtaining access to the original document object.

```
<script>
  "use strict";
  ADSAFE.id("test");
</script>

<div id="test">
  <script>
    "use strict";
    % set adsafe_lib.___nodes___ to an untrusted (malicious) object.
    ADSAFE.lib("___nodes___",
               function(lib){
                 var o = [{appendChild: function(x) {var steal = x.ownerDocument)},
                           tagName:1}];
                 return o;});

    ADSAFE.go("test",
               function(dom,lib){
                 % lib points to the adsafe_lib object.
                 var frag = dom.fragment();
                 var f = frag.value;
                 % f now points to the value method of the dom library.
                 lib.v = f;
                 lib.v();
               });
  </script>
</div>
```

Figure 8.8: ADSafe$_{Nov10}$ exploit code

Fixing the Attack. A fix for the attack is to rewrite the lib method using the annotation $Safe in the following way.

```
function (name, f) {if(!reject_name(name)) {adsafe_lib[name, $Safe] = f(adsafe_lib);}
```

With this rewriting, ENCAP reports no DOM leaks, thus proving that the ADSafe_{Nov09} API safely confines DOM objects under the added annotations and the SES-light threat model. We reported the vulnerability to Yahoo! and the corresponding fix was adopted immediately.

8.3.2 Sealer-Unsealer Pairs

Our next example is an implementation of the Sealer-Unsealer encapsulation technique, which was first introduced by Morris [38] in 1973, for providing an encryption-decryption like mechanism for functions. Sealer-Unsealer pairs are important security mechanisms used in designing capability based systems. We analyzed the SES-light implementation of sealers and unsealers, shown in Figure 8.9. The API exposed to untrusted code provides access to the method "seal" and a sealed function secret . By running ENCAP on the implementation we successfully verified that the API confines the secret function.

8.3.3 Mint

Our final example is the Mint function, which is a canonical example used in the object-capability literature to demonstrate how capability patterns like sealers and unsealers can be used for writing robust code that can be safely run in potentially malicious environments. An SES-light implementation of Mint is present in Figure 8.10. Untrusted code is handed the function Mint that provides a reference to the Purse constructor. The Purse constructor can be invoked to create a *purse* object which encapsulates a field named "balance" that stores the purse's balance, and has methods "deposit" and "getBalance"" to respectively read and update the "balance" field. One of the correctness goals for Mint is *conservation of currency*, which says that the sum of balances of all purse objects must be constant. A quick inspection of

```
function SealerUnsealer() {
  var flag = false;
  var payload = null;
  return {seal: function(payloadToSeal) {
                  return function() {
                           flag = true;
                           payload = payloadToSeal;
                         }
                },
          unseal: function(box){
                    flag = false;
                    payload = null;
                    try{
                      box();
                      if (!flag)
                        throw "Invalid Box";
                      return payload;
                    } finally{
                     flag = false;
                     payload = null;
                    }
                  }};
}

% define a function secret.
function Secret() { }

% seal the secret function.
var brand = SealerUnsealer();
var box = brand.seal(Secret);

% expose the sealer and the sealed function to untrusted code.
var api = {seal: brand.seal, sealedFunc: box}
```

Figure 8.9: Implementation of Sealer-Unsealer pairs in SES-light

```
function Nat(n) { if (n !== n >>> 0) { throw "Not Natural"; } return n; }

function Mint() {
  var brand = SealerUnsealer();
  return function Purse(balance){
          function decr(amount){
            balance = Nat(balance − amount);
          }
          return {getBalance: function() {return balance;},
                  makePurse: function() {return Purse(0);},
                  getDecr: function() {return brand.seal(decr);},
                  deposit: function(amount,src) {
                             var box = src.getDecr();
                             var decr = brand.unseal(box);
                             Nat(balance + amount);
                             decr(Nat(amount));
                             balance += amount;
                           }};
        };
}

% expose the Mint function to untrusted code.
var api = Mint;
```

Figure 8.10: Implementation of the Mint in SES-light

the code reveals that the function "decr" can directly alter the "balance" field. Thus conservation of currency necessitates that the Mint object must *safely confine* the "decr" function. By running `ENCAP` on the code in Figure 8.10, combined with the implementation of sealer-unselars pairs from Figure 8.9, we successfully verified that the decr function is safely confined, under the SES-light threat model.

8.3.4 Summary

We demonstrated the effectiveness of our tool `ENCAP` by using it to find a security-oversight in the Yahoo! ADsafe library, and then verifying confinement of the repaired library and other benchmark examples from the object-capability literature.

The vulnerability that ENCAP found in the ADsafe library is not only exploitable using untrusted SES-light code, but also using code that satisfies the stronger JSLint syntactic restrictions imposed by ADSafe_{Nov10}. The exploit code in Figure 8.8, when added to a hosting page that contains the (broken) ADsafe library, is able to obtain a reference to the document object. We successfully tested this exploit on Firefox 3.5.12, Chrome 2.1 and Safari 4.1.1. Although ADsafe has been subject to much previous scrutiny, including prior automated static analyses [40], this attack had not previously been discovered.

Perhaps surprisingly, there also exist examples of APIs that enforce confinement under standard ES3 semantics but not under SES-light semantics. For example, the following API fails to confine the function critical under the SES-light semantics, but does so under the ES3 semantics.

```
var x = function critical() { };
var api = function() {var a = this;
            if (typeof a === "object")
              delete a.x;
            return x;
          }
```

This is because in the ES3 semantics, the this value of the api function would be the global object and therefore the priv binding would get deleted before the return step. However under the SES-light semantics, the this value would be undefined thereby making the function return critical.

Finally, we note that ENCAP has the expected limitations and imprecisions associated with flow insensitive and context insensitive analysis. For instance, running ENCAP on the Cajita run-time library of the Google Caja framework [11], generated a large number of false positives due to the context-insensitivity of the analysis. We believe that further technical work on analysis methods for SES-light can help eliminate some of the present limitations, thereby aiding future programmers in developing security-critical code. For example, additional analysis methods such as object-sensitive analysis [57] and CFA2 techniques [85] may be used to increasing the

precision of the analysis, and also provide more useful diagnostics when confinement cannot be established.

8.4 Related Work

There is a long history of using static analysis and language-based techniques to secure extensible software, including such notable work as Typed Assembly Language [62], Proof-Carrying Code and Software-based Fault Isolation [87]. However, this line of research has focused on providing strong guarantees about untrusted extensions, and their access to trusted interfaces to security-relevant services. Less considered have been the effects of giving an arbitrary, untrusted extension unfettered access to such trusted interfaces. Until recently, most work that considered such "API security" had centered around cryptographic security modules, and their interfaces [9]. For those cryptographic APIs, keys take the role of security-critical objects, and static analysis has been used to establish whether (or not) those keys are properly confined within the security module. This line of work has strong connections to formalisms such as BAN logic [10], where similar abstract analysis can be used to reason about all possible interactions in security protocols. As security-relevant services that expose rich interfaces are increasingly written in high-level, type-safe languages, such abstract analysis of the security properties of APIs has increasingly wider applicability.

For server-side Web software written in languages other than JavaScript, several efforts have employed static analysis for security, in particular to identify and prevent Cross-Site Scripting (CSS) attacks or SQL injection. Examples include the taint-based CSS analysis in Pixy [37], the SQL injection analysis by Xie and Aiken [92], both in the context of PHP. In addition, in the context of Java, Livshits and Lam implemented a Datalog-based analysis to establish security properties such as proper sanitization [43]. Compared to this work, JavaScript raises unique challenges, in particular due to its highly-dynamic nature.

Recently, flow-insensitive static analysis of JavaScript code has been considered in the research efforts Staged Information Flow [13] and Gatekeeper [25]. Both efforts make use of mostly-static techniques, supported by some runtime checks; in particular,

Staged Information Flow leaves to runtime checks the analysis of all dynamic code and *eval*. Gatekeeper has perhaps the most similar goals to our work: it aims to constrain potentially-obfuscated, malicious JavaScript widgets that execute within a host Web page, and invoke the APIs of that Web page. Gatekeeper analysis also makes use of Datalog, in much the same way as we do in our work. Gatekeeper, however, does not statically analyze eval and does not provide a rigorous proof of soundness for their analysis. As a final point of comparison, the VEX system uses static information-flow analysis to find security vulnerabilities in Web browser extensions. Much like in the current work, VEX analysis is based on a formal semantics for a fragment of JavaScript, based on [27, 46]. Despite several similarities, VEX is fundamentally different from the current work in both its application domain, and in its technical details. VEX aims to prevent script injection attacks, and analyzes only certain types of explicit flows from untrusted sources to executable sinks; in comparison, we consider the confinement of security-critical objects. The static analysis in VEX is path-sensitive, context-sensitive and makes use of precise summaries. but is fundamentally unsound. In comparison, our static analysis is simpler, applies to the core of an important new JavaScript variant, and guarantees soundness.

Chapter 9

Conclusion

JavaScript has come a long way since its birth in 1995. Originally designed for adding small scripting capabilities to Web pages, it has now evolved into becoming the "assembly language" for large applications such as GMail, Google Maps, Facebook, and many others. JavaScript makes it particularly easy for code to compose and as a result most Web applications today are mashups obtained by combining JavaScript content from multiple sources. Since third-party sources can be untrusted, most Web pages use browser-based or language-based methods to keep such content from maliciously altering pages, stealing sensitive information, or causing other harm. There are a number of such JavaScript "sandboxing" mechanisms, including Facebook FBJS, Yahoo! ADSafe, Google Caja, and Microsoft Web Sandbox. Despite their popularity, these mechanisms do not come with any rigorous specifications or guarantees. This dissertation supports the thesis that it is possible to design provably-correct and practical language-based mechanisms for sandboxing third-party JavaScript on Web pages.

When we started this work in early 2008, it was unclear to us as to what security policies must be enforced by JavaScript sandboxing mechanisms. In order to understand this we analyzed two prominent real-world JavaScript sandboxing mechanisms: Facebook FBJS and Yahoo! ADsafe. Both these mechanisms were designed by engineers at large software companies with the goal of sandboxing JavaScript in

the "wild". We reverse-engineered the implementations of these mechanisms and discovered their main security goals as: (i) mediating access to certain sensitive DOM objects, and (ii) isolating one third-party component from another. Our original objective was to prove that these goals are correctly enforced by FBJS and ADsafe. However while setting up the proof we identified multiple exploitable vulnerabilities in both these mechanisms. Instead of being simple oversights on part of the developer, these vulnerabilities were actually quite subtle and stemmed from certain non-standard features of JavaScript such as the ability for code to obtain a direct reference to its scope, the ability to redefine built-in objects, and implicit conversions of objects to primitive values. Moreover we found that the semantics of certain JavaScript constructs such as the `this` argument in function calls, named recursive functions, and `try-catch` statements were quite complex and misunderstood by many JavaScript developers. In order to have a clear understanding of the semantics of JavaScript and to set up a formal foundation for reasoning about JavaScript sandboxing mechanisms, we developed a small-step operational semantics for JavaScript based on the 3^{rd} edition of the ECMA-262 standard (see Chapter 4). The standardized language is called ECMAScript3 (ES3).

The rest of this research focussed on using the semantics to both systematically design and prove the correctness of sandboxing mechanisms for JavaScript. In designing our mechanisms we adopted the API+Language-based-sandboxing that is used by FBJS and ADsafe. The crux of this architecture is to have a trusted API that mediates all access to critical hosting page resources and a language-based sandboxing mechanism for restricting third-party code so that it can *only* access the API and cannot interfere with the execution of other third-party components. An attractive feature of this architecture is that modifying the access policy on critical resources only requires modifying the API, the language-based sandbox stays the same. In light of this feature, we formalized two problems that need to be solved while designing mechanisms based on this architecture — the *Sandbox Design* problem and the *API Confinement* problem. The Sandbox Design problem involves designing a language-based sandbox for restricting third-party code to an API, and isolating one

third-party component from another. The API Confinement problem requires verifying that a given API implementation safely confines critical hosting page resources, that is, the API ensures that no sandboxed third-party code can use it to obtain a direct reference to a critical hosting page resource.

Armed with the insights gained from our operational semantics for ES3, we designed filtering and source-to-source rewriting based techniques for solving the Sandbox Design problem (see Chapter 5 and Chapter 6). Borrowing ideas from the literature on capability-based protection, we developed the concept of *Authority-Safety* in order to design a mechanism for isolating one third-party component from another. We backed our mechanisms with rigorous proofs of correctness carried out manually using our operational semantics. We compared our mechanisms with FBJS and showed them to be as permissive as FBJS. In addition, we also found new previously undiscovered vulnerabilities in both FBJS and ADsafe. We reported these vulnerabilities to the concerned developers along with suitable fixes which in most cases were promptly adopted. While our mechanisms are very close to current FBJS, we consider it a success that we were able to contribute to the security of Facebook through insights obtained by our semantic methods, and that in the end we are able to provide provable guarantees for a JavaScript sandboxing mechanism that is essentially similar to one used by external application developers for a hugely popular current site.

We initially attempted to solve the API Confinement problem for APIs written in ES3. However the attempt failed due to the various dynamic features of ES3 that make static analysis very challenging. Fortunately, in December 2009, the ECMA Standards committee released a 5^{th} edition[1] of the JavaScript standard along with an additional "strict mode". While the restrictions imposed by the strict mode eliminated a number of ES3 limitations, two issues still remained: dynamic code generation and mutability of built-ins. To address these issues we defined a sub-language SES of ES5-strict that gets rid of both these issues and is amenable to static analysis and defensive programming. The language SES has been under proposal by the EMCA Standards committee (TC39) for adoption within future versions of JavaScript.

We defined a small-step operational semantics for the core of SES, which we call

[1]due to disagreements within the ECMA committee the 4^{th} edition never got released.

SES-light (see Chapter 7). Using the semantics as a foundation, we developed a provably-sound static analysis technique for proving confinement of SES-light APIs (see Chapter 8). We implemented the technique in the form of an automated tool `ENCAP` and used it find a previously undetected confinement vulnerability in the Yahoo! ADsafe DOM API. We subsequently proved confinement for a repaired version of the API, and also demonstrated confinement for other isolation examples from the object-capability and security literatures. While our results apply to SES-light only it is possible to extend the techniques to full SES.

In summary, the above mentioned results establish that it is possible to systematically design practical and provably-correct sandboxing mechanisms for JavaScript. Furthermore, they also demonstrate that a formal operational semantics is immensely useful in both designing and analyzing the correctness of language-based sandboxing mechanisms.

9.1 Perspective

Despite its immense popularity, JavaScript based on 3^{rd} edition of the ECMA-262 standard is considered a notoriously difficult language for writing secure, correct and efficient code. The key reason behind this is the extreme dynamism of the language which makes it impossible to design type systems, program analyses and program optimizations. Although certain features such as dynamic typing and mutable object properties add value in supporting a flexible programming model and lowering the learning curve for novice programmers, other features such as the ability to freely manipulate scope objects pose significant hurdles to both developers and code analysis tool designers. Based on our experience on designing sandboxing mechanisms for ES3 and our interactions with the ECMA Standards committee (TC39), we identified five key limitations of ES3 that make static analysis and defensive programming challenging.

(1) **Lack of lexical scoping:** Due to the presence of prototype chains on scope objects, it is possible for ES3 code to manipulate variables outside their own scope, thereby violating lexical scoping.

(2) **Lack of closure-based encapsulation:** Most browser implementations of ES3 support the "caller" property that allows callee code with a mechanism to access its caller function, thereby breaking closure-based encapsulation.

(3) **Ambient access to the global object:** ES3 provides multiples channels for code to obtain a direct reference to the global scope object, thereby making it difficult to isolate code.

(4) **Mutable built-in objects:** Almost all properties of all built-in objects, including the global object, can be manipulated at runtime. As a result trusted code cannot rely on built-in objects in the presence of untrusted extensions.

(5) **Dynamic code generation:** A fundamental static analysis hurdle in ES3 is the presence of constructs like eval that allow code to get generated at run-time and therefore outside the scope of static reasoning.

Recognizing the first three limitations, the ECMA Standards committee ensured that the "strict mode" in the 5^{th} edition of the standard supports lexical scoping, safe closures and an isolated global object. Our sub-language SES eliminates the remaining limitations by requiring all built-in objects to be immutable, and all dynamically generated code to be local to the containing scope. Although SES requires programmers of trusted code to use a more limited form of JavaScript, we believe that its clean semantic properties and the power of `ENCAP` and other analysis methods enabled by it would provide ample motivation for concerned programmers to adopt this language. In fact, the success of our tool on some existing code suggests that careful programmers may already respect some of the semantically motivated limitations of SES. Furthermore, we believe that the SES restrictions make JavaScript very similar to standard dynamically typed functional programming languages, and the techniques developed for analyzing such languages would be applicable to SES as well. Finally, we remark that the API+Language-based-sandboxing architecture provides a promising approach to sandboxing JavaScript code, and also code written in other similar scripting languages that only support limited forms of static analysis. The governing design principle behind this architecture is to package all security-critical

resources behind an API and then allow untrusted third-party code to freely invoke the methods on the API. The foundation of the design lies in the *object-capability* theory of securing systems (see [58, 42]). The methods on the API are effectively *capabilities* supplied to third-party code and the sandboxing mechanism is the *loader* that loads third-party code with the given set of capabilities. This design does not require any sophisticated static analysis of third-party code except for the check that it cannot access any global state outside of the API. It does require static analysis of the trusted API implementation for confinement, however given the static analysis challenges associated with arbitrary JavaScript this is a favorable tradeoff. This is because it only requires defining the trusted API implementation in a restricted fragment of JavaScript that is amenable to static analysis, while allows third-party code to freely make use of full JavaScript or a minimally restricted fragment of it.

Appendix A

Proofs

A.1 Proofs from Chapter 4

In this section we prove Theorem 1 from Chapter 4.

A.1.1 Preliminaries

We first extend the definition of well-formedness of terms to contexts. Given a well-formed heap H, $Wf_{con}(C, H)$ holds for an evaluation context C if C is derivable from the context grammar (see [45]) and $\forall l : l \in C \implies l \in dom(H)$.

A.1.2 Main Results

We first prove a lemma that states that well-formedness of terms is compositional.

Lemma 1 *For all well-formed heaps H, evaluations contexts C and appropriate terms t, $Wf_t(C[t], H)$ holds iff $Wf_{con}(C, H) \wedge Wf_t(t, H)$.*

Proof Sketch: In order to prove the left-to-right direction we start by assuming $Wf_t(C[t], H)$. From the definition of Wf_t it follows that $C[t]$ is derivable from the user grammar and $\forall l : l \in C[t] \implies l \in dom(H)$. As a result C is derivable from the context grammar and $\forall : l \in C \implies l \in dom(H)$. Hence $Wf_{con}(H, C)$ holds. For the right-to-left direction we first prove by structural induction that if a term t is derivable

from the term grammar and context C is derivable from the context grammar then $C[t]$ is derivable from the term grammar. The lemma follows immediately from this result. □

We now prove Theorem 1

Restatement of Theorem 1. *For all states S_1 such that $Wf(S_1)$ holds:*

- *(Preservation) If there exists a state S_2 such that $S_1 \to S_2$, then $Wf(S_2)$ holds.*
- *(Progress) If $term(S_1) \notin Vals$ then there exists a state S_2 such that $S_1 \to S_2$.*

Proof Sketch: We prove the Preservation theorem by proving the following stronger property.

$$Wf(H_1, L_1, t_1) \wedge H_1, L_1, t_1 \to H_2, L_2, t_2 \Longrightarrow Wf(H_2, L_2, t_2) \wedge dom(H_1) \subseteq dom(H_2)$$

We prove this property by an induction over the set of reduction rules. Transition axioms form the base case and the contextual rules form the inductive case. For the transition axioms we show the property by a case-by-case analysis. As an example, we give the argument for the following axiom:

$$\frac{Scope(H, L, \text{“x”}) = ln}{H, L, x \xrightarrow{e} H, L, ln\text{*“x”}}$$

If $Wf(H, L, \text{“x”})$ holds then by definition the heap H is well formed and the stack L is well formed with respect to the heap. Therefore $\forall l : l \in t \implies l \in dom(H)$. For the definition of the *Scope* it follows that for all H, L, m and identifiers $Scope(H, L, m) =$ null or $Scope(H, L, m) \in L$. As a result $Wf_t(ln\text{*“x”}, H)$ holds and the final state $H, L, ln\text{*“x”}$ is also well formed.

In order to prove the inductive case, we must show that for any contextual rule of the form

$$\frac{H_1, L_1, t_1 \to H_2, L_2, t_2}{H_1, L_1, C[t_1] \to H_2, L_2, C[t_2]}$$

if $Wf(H_1, L_1, C[t_1])$ holds then $Wf(H_2, L_2, C[t_2])$ holds and $dom(H_1) \subseteq dom(H_2)$. From the definition of Wf and Lemma 1 it follows that

$$Wf(H_1, L_1, C[t_1]) \implies Wf(H_1, L_1, t_1) \wedge Wf_{con}(H_1, C) \tag{A.1}$$

From the induction hypothesis we have

$$Wf(H_1, L_1, t_1) \implies Wf(H_2, L_2, t_2) \wedge dom(H_1) \subseteq dom(H_2) \tag{A.2}$$

From the definition of well-formedness of term contexts we have

$$Wf(H_1, C) \wedge dom(H_1) \subseteq dom(H_2) \implies Wf(H_2, C) \tag{A.3}$$

Combining conditions A.1, A.2, and A.3 it follows that

$$Wf(H_1, L_1, C[t_1]) \implies \left(\begin{array}{c} Wf(H_2, L_2, t_2) \wedge Wf_{con}(H_2, C) \\ \wedge \ dom(H_1) \subseteq dom(H_2) \end{array} \right) \tag{A.4}$$

Combining the above condition with Lemma 1 and the definition of Wf, it follows that $Wf(H_1, L_1, C[t_1]) \implies Wf(H_2, L_2, C[t_2]) \wedge dom(H_1) \subseteq dom(H_2)$. Hence the inductive case holds.

The Progress theorem can be proven by a structural induction over the terms. For the base cases we show that they are either values or exceptions or have a transition axiom that applies to them. For the inductive case, we show that for each expression, statement and program there is either a transition axiom or a context rule that applies, or the term is a value or an exception, in which case the theorem is directly true. As an example consider the expression e_1=e_2. If e_1 is not a value then the contextual rule for evaluating expressions applies for the context $\_ = e_2$. If e_1 is a value v and e_2 is not a pure value then the contextual rule for evaluating expressions applies for the context $v = \_$. Finally, if e_1 is a value v and e_2 is a pure value va then the transition axiom for the assignment expression $v = va$ applies. □

A.2 Proofs from Chapter 5

In this section we prove Theorem 3 from Chapter 5. Our proof is based on the formal framework developed in Section 4.4.

A.2.1 Preliminaries

In order to prove Theorem 3 we must show that for all third-party terms $t \in \mathit{Terms}^{ES3}_{user}$ and global variable whitelists $\mathcal{G}$ such that $\mathcal{P}_{nat} \subseteq \mathcal{G}$, $\mathit{Isolation}^h_{\mathcal{G}}(H^h, L^h, \phi^h(t))$ holds. To prove this theorem we first define a property Safe^h on program states and show that if Safe^h holds for all states appearing on the execution trace of a state H, L, t then $\mathit{Isolation}^h_{\mathcal{G}}(H^h, L^h, \phi^h(t), \mathcal{G})$ holds.

Definition 23 *(Safe^h) Given a state $S := H, L, t$; $\mathit{Safe}^h(S)$ holds iff $\mathcal{N}(t) \subseteq \mathcal{G}$ and $\forall a : (a \in \mathit{Act}(H, L, t) \ \wedge \ \mathit{loc}(a) = \#\mathit{global}) \implies \mathit{props}(a) \in \mathcal{N}(t) \ \cup \ \mathcal{P}_{nat}$.*

Given a state $S := H, L, t$ and an evaluation context C (see [45] for the complete context grammar), we define $C[S]$ as the state $H, L, C[t]$.

Proposition 1 *For all states S and evaluation contexts C, $\mathit{Safe}^h(S) \Longrightarrow \mathit{Safe}^h(C[S])$.*

Proof Sketch: The proposition holds trivially for all non-well-formed states. For a well-formed state S, from the semantics of contextual rules we have that, $\mathit{Act}(C[S]) = \mathit{Act}(S)$. The proposition immediately follows from this property. □

The property Safe^h is naturally extended to traces τ by defining $\mathit{Safe}^h(\tau) := \forall S : S \in \tau \implies \mathit{Safe}^h(\tau)$. We now show that if the safety holds for a reduction trace of a state, then hosting page isolation holds for the state.

Lemma 2 *For all whitelists $\mathcal{G}$ such that $\mathcal{P}_{nat} \subseteq \mathcal{G}$, $\forall H, L, t : \mathit{Safe}^h(\tau(H, L, t)) \implies \mathit{Isolation}^h_{\mathcal{G}}(H, L, t)$.*

Proof Sketch: Given a well-formed state H, L, t, from Theorem 1, we have that all states S in $\tau(H, L, t)$ are also well-formed and satisfy $\mathcal{N}(\mathit{term}(S)) \subseteq \mathcal{G}$. Furthermore, $\mathit{Act}(\tau(H, L, t)) = \bigcup_{S \in \tau(H,L,t)} \mathit{Act}(S)$. The lemma now follows immediately from the definitions of Safe^h and $\mathit{Isolation}^h_{\mathcal{G}}$. □

In the rest of this section we prove that $Safe^h(\tau(H^h, L^h, \phi^h(t)))$ holds for all terms t in $Terms^{ES3}_{user}$. The proof involves defining a *goodness* property $Good^h$ on states and showing that the following hold: (1) (*Initial Goodness*) All states S in the set $Init^h = \{H^h, L^h, \phi^h(t) \mid t \in Terms^{ES3}_{user}\}$ are good, and (2) (*Goodness Preservation*) For any non-final good state S_1 , there exists a good state S_2 such that $S_1 \rightsquigarrow S_2$, all states S in the sub-trace $subTr(S_1, S_2)$ are safe.

In order to define $Good^h(S)$, we first step up the following notations and definitions. We use the following notations for certain important heap locations. l_g, $l_{Function}$, l_{String}, l_{eval} are the heap locations of the global object, the constructors @Function, @String, and function @eval respectively. l_{OP} and l_{AP} are the heap locations of the built-in Object.prototype and Array.prototype objects respectively. $l_{valueOf}$ is location of the "valueOf" method of Object.prototype on the initial heap H_0. l_{sort}, l_{concat}, $l_{reverse}$ are the locations of the methods "sort", "concat" and "reverse" of Array.prototype on the initial heap H_0. l_{hop}, l_{pie}, l_{call} are the locations of the methods "hasOwnProperty" and "propertyIsEnumerable" of Object.prototype, and method "call" of Function.prototype on the initial heap H_0. $l_{valueOfN}, l_{sortN}$, $l_{concatN}$ and $l_{reverseN}$ are the locations of the wrapper objects on the heap H^h, that get created by the initialization codes $T_{valueOf}$, T_{sort}, T_{concat}, and $T_{reverse}$ respectively. l_{idx} and l_{ng} are respectively the heap locations of "$idx" and "ng" methods of Object.prototype created by the initialization codes T_{idx} and T_{ng}. Finally, we define
F_{idx} := function(){return (x=({ }).$String(x),CHECK[x])} as the function expression returned by the function at location l_{idx}.

In order to define the property $Good^h(S)$, we first define two auxiliary properties: *term goodness* $TGood^h(t)$ for terms t, and *heap goodness* $HGood^h(H)$ for heaps H.

Term goodness $TGood^h(t)$**.** : Term goodness is defined as a conjunction of a set of simple syntactic constraints on the structure of the term. While is possible to formally define term goodness by defining a grammar for good terms, we choose to define it simply by stating the syntactic constraints. This helps in avoiding a lot of notational overhead. Let $\mathcal{B}$ be the union of the set of names {"eval", "Function", "constructor"} and the set of all names beginning with the symbol '$'. A well-formed term t is good iff it satisfies the following:

(1) All identifiers x appearing in t are named in the set "x" $\in \mathcal{G} \setminus \mathcal{B}$.

(2) Structure of t does not contain any explicit property name (x) from the blacklist $\mathcal{B}$ except inside the context ({})._(e). The only blacklisted property names that can appear within the context ({})._(e) are $ng and $idx.

(3) All sub-expressions of t of the form l*m and @AddProps(m, e, l, {[$p\tilde{n}$:e]}) must satisfy $m \notin \mathcal{B}$. Furthermore, for the expression l*m, if $l = l_g$ then $m \in \mathcal{G}$.

(4) If the term contains a sub-expression $e_1[e_2]$ then e_2 must be of one of the following: (A) IDX(e) for some expression e such that $TGood^h(e)$. (B) String m such that $m \notin \mathcal{B}$.

(5) If t contains this then it must appear only inside the context ({}).$ng(_).

(6) Structure of t does not contain any @cEval or @FunParse sub-terms.

(7) Structure of t does not contain any heap addresses from the set $\{l_{Function}, l_{eval}\}$ $\cup$ $\{l_{valueOf}, l_{sort}, l_{concat}, l_{reverse}\}$.

(8) If the heap address of the global object l_g is present in t then it must appear inside one of the following contexts only: Function(fun([$\tilde{x}$]){P},_); _.@Put(m,va); ł.@call(_,[$\tilde{va}$]); l.@exe(_,[$\tilde{va}$]); @Fun(_,e[$\tilde{va}$]); @ExeFPA(l,_,va); @FunExe(_,P); _*m

Heap goodness ($HGood^h(H)$). : We now state the goodness property for heaps. A heap H is good iff the following properties hold.

(1) $\forall l, p : H(l).p = l_{Function} \implies p =$ "constructor" $\vee$ $p =$ "Function"

(2) $\forall l, p : H(l).p = l_{eval} \implies p =$ "eval"

(3) $\forall l, p : H(l).p = \#global \implies p =$ @this $\vee$ $p =$ @Fscope

(4) $\forall l, p : p \in H(l) \wedge isPrefix(\text{'\$'}, p) \implies l = l_{OP}$

(5) $\forall l, p : H(l).p = l_{valueOf} \implies l = l_{OP} \wedge p =$ "$OPvalueOf"

(6) $\forall l, p : H(l).p = l_{concat} \implies l = l_{OP} \wedge p =$ "$APconcat"

(7) $\forall l, p : H(l).p = l_{sort} \implies l = l_{OP} \wedge p =$ "$APsort"

(8) $\forall l, p : H(l).p = l_{reverse} \implies l = l_{OP} \wedge p =$ "$APreverse"

(9) $H(l_{OP}).$"$g" $= \#global$

(10) $H(l_{OP}).$"$String" $= l_{String}$

(11) $H(l_{valueOf}).$"call" $= l_{call}$

(12) $H(l_{valueOfN}).$@body = function(){var $ = ({}).$OPvalueOf.call(this, arguments);
return ($==({}).$g? null: $) }

(13) $H(l_{valueOfN}).$"@Fscope" $= \#global$

(14) $H(l_{sort}).$"call $= l_{call}$

(15) $H(l_{sortN}).$@body = function(){var $ = ({}).$APsort.call(this, arguments);
return ($==({}).$g? null: $) }

(16) $H(l_{sortN}).$@Fscope $= \#global$

(17) $H(l_{reverse}).$"call $= l_{call}$

(18) $H(l_{reverseN}).$@body = function(){var $ = ({}).$APreverse.call(this, arguments);
return ($==({}).$g? null: $) }

(19) $H(l_{reverseN}).$@Fscope $= \#global$

(20) $H(l_{concat}).$"call" $= l_{call}$

(21) $H(l_{concatN}).$@body = function(){var $ = ({}).$APconcat.call(this, arguments);
return ($==({}).$g? null: $) }

(22) $H(l_{concatN}).$@FScope $= \#global$

(23) $H(l_{OP}).$"$ng" $= l_{ng}$

(24) $H(l_{ng})$.@body = function(x){ x== ({ }).$g? null: x)}

(25) $H(l_{OP})$."$idx" $= l_{idx}$

(26) $H(l_{OP})$.@body = function(x){ return function(){

return (x=({ }).$String(x),CHECK[x])}};

(27) $\forall l : l \notin L_{wrapped} \wedge$ @body $\in H(l) \implies TGood^h(H(l).\text{@body}) \vee H(l).\text{@body} = F_{idx}$

where $L_{wrapped} := \{l_{valueOfN}, l_{reverseN}, l_{sortN}, l_{concatN}, l_{ng}, l_{idx}\}$

Definition 24 *(State Goodness $Good^h$) Given a state (H, L, t), $Good^h(H, L, t)$ holds iff $Wf(H, L, t)$, $HGood^h(H)$ and $TGood^h(t)$ hold.*

A.2.2 Main Results

We begin by proving Theorem 2 that states that $\mathcal{S}^h$ is a well-formed sandboxing mechanism.

Restatement of Theorem 2. *$\mathcal{S}^h$ is a well-formed single-component sandboxing mechanism.*

Proof Sketch: To prove this theorem, we show that $Wf_h(H^h) \wedge Wf_s(L^h, H^h)$ holds and for all terms $t \in Terms^{ES3}_{user}$, $Wf_t(t, H^h)$ implies $Wf_t(\phi^h(t), H^h)$. By definition, H^h, L^h are defined as the final heap and stack obtained on executing the term $T_{in} := T_{ng}; T_{valueOf}; T_{sort}; T_{concat}; T_{reverse}; T_{idx}$. Since the term T_{in} is well formed, it follows from Theorem 1 (preservation part) that H^h, L^h are well formed. Therefore $Wf_h(H^h) \wedge Wf_s(L^h, H^h)$ holds. From the definition of ϕ^h and the grammar for user-level ES3 terms (see figures 4.2 and 4.3), it follows that if $t \in Terms^{ES3}_{user}$ then $\phi^h(t) \in Terms^{ES3}_{user}$. As a consequence, $Wf_t(t, H^h)$ implies $Wf_t(\phi^h(t), H^h)$. □

We now move to the proof of Theorem 3 We begin by formally proving the properties *Initial Goodness* and *Goodness Preservation*) defined in the beginning of this section.

Lemma 3 *(*Init*) All states in the set* $\{H^h, L^h, \phi^h(t) \mid t \in Terms^{ES3}_{user}\}$ *are good.*

Proof Sketch: By definition,
$H^h := heap(final(H^{ES3}_0, L^{ES3}_0, T_{in})$ and $L^h := stack(final(H^{ES3}_0, L^{ES3}_0, T_{in})$ where $T_{in} := T_{ng}; T_{valueOf}; T_{sort}; T_{concat}; T_{reverse}; T_{idx}$. Furthermore, the state $H^{ES3}_0, L^{ES3}_0, T_{in}$ is always well formed. $HGood^h(H^h)$ can therefore be shown by symbolically executing the term T_{in} and tracking all heap updates, and then show that the final heap obtained is good.

For all terms $t \in Terms^{ES3}_{user}$, $TGood^h(\phi^h(t))$ can be show by induction over the structure of terms. All cases are straightforward except the inductive case for $e_1[e_2]$ which we discuss next. By definition, $\phi^h(e_1[e_2])$ is either 0 or $\phi^h(e_1)[\mathsf{IDX}(\phi^h(e_2))]$. $TGood^h(0)$ holds trivially. By the hypothesis, $TGood^h(\phi^h(e_1))$ and $TGood^h(\phi^h(e_2))$ hold. From the definition of term goodness it follows that $TGood^h(\phi^h(e_1)[\mathsf{IDX}(\phi^h(e_2))])$ holds. This completes the inductive case for $e_1[e_2]$. □

In order to prove the theorem, we first prove that term goodness is compositional.

Proposition 2 *For all terms t; and all evaluation contexts C except the context va[_]; if C[t] is good then the following hold:*

- *t is good.*
- $\forall t : TGood^h(t) \implies TGood^h(C[t])$.

Proof Sketch: We prove this proposition by a case analysis on the set of all evaluation contexts. As an example we present the case for the context $va[\_]$. From the definition of $TGood^h$, it follows that for any expression e, $TGood^h(va[e])$ holds iff (1) $va \notin \{l_{Function}, l_{eval}\} \cup \{l_{valueOf}, l_{sort}, l_{concat}, l_{reverse}\} safeLoc(va)$ holds, and (2) e is either of the form $\mathsf{IDX}(e_2)$ for some good expression e_2 or a string m such that $m \notin \{\text{“eval”}, \text{“Function”}, \text{“constructor”}\} \wedge \neg isPrefix(\text{‘\$’}, p)$ $safeProp(m)$ holds. From the definition of IDX macro it follows that if expression e_2 is good then $\mathsf{IDX}(e_2)$ is also good. □

We now prove our main lemmas about goodness preservation. Given a reduction rule R from our semantics $(\Sigma, \mathcal{R}^{ES3}_{frozen})$, let $\mathcal{S}_R$ be the set of states that can be reduced using the rule R. By definition all states in $\mathcal{S}_R$ are non-final.

Proposition 3 *For all good states $S_1 \in \mathcal{S}_{[I-Call]}$, without loss of generality $term(S_1)$ can be assumed to be of the form l_1.@Call(l_2, [$\tilde{va}$]). The following properties hold*

(1) *If $l_1 \in L_{wrapped}$ then S_1 always evaluates to a good final state S_v such that $Safe^h(\tau(S_1))$ holds.*

(2) *If $l_1 \notin L_{wrapped}$ and $heap(S_1)(l_1)$.@body $= F_{idx}$ (see definition of heap goodness) then S_1 always evaluates to a good final state S_v such that $Safe^h(\tau(S_1))$ holds and $termS_v \notin \mathcal{B}$ where $\mathcal{B}$ is the property name blacklist.*

(3) *If $l_1 \notin L_{wrapped}$ and $heap(S_1)(l_1)$.@body $\neq F_{idx}$ then the S_1 reduces to a good state S_2 in a single step and $Safe^h(S_1)$ holds.*

where $L_{wrapped} := \{l_{valueOfN}, l_{sortN}, l_{concatN}, l_{reverseN}, l_{ng}, l_{idx}\}$.

Proof Sketch: Since the state S_1 is good, the heap locations in $L_{wrapped}$ store the various wrapper functions defined by the initialization code. These functions do not contain any recursion or loops, and therefore the corresponding function calls terminate in finite steps. We prove the first property by symbolically executing the functions and showing that all states in the resulting traces are safe and that the final heaps and values obtained area always good.

For the second property, we argue exactly like the first case, that is, using symbolic execution. The body of the function expression F_{idx} includes the closure variable x. For soundness, we keep this variable symbolic and therefore prove the result for all possible values that x can resolve to.

For the third property, since S_1 is good, we have that $H(l_1)$.@body is a good term. From the definition of the transition axiom I-Call it is easy to check that the state obtained after a single step reduction is good, and that all heap access performed during the reduction are safe. Therefore $Safe^h(S_1)$ holds. □

Lemma 4 *For any non-final good state S_1, there exists a good state S_2 such that $S_1 \rightsquigarrow S_2$, and $Safe^h(subTr(S_1, S_2))$ holds.*

Proof Sketch: We prove this lemma by an induction over the set of reduction rules $\mathcal{R}^{ES3}$ We show that for all rules R, for all good states $S_1 \in \mathcal{S}_R$, there exists a good state S_2 such that $S_1 \rightsquigarrow S_2$, and $Safe^h(subTr(S_1, S_2))$ holds.

Base case.The base case is the set of transition axioms. We prove the base case by case analysis. For the axiom [I-Call], the proof follows from Proposition 3. For all axioms R, except the function call axiom ([I-Call]), we show that for any good state S_1 in $\mathcal{S}_R$, $Safe^h(S_1)$ holds and the state S_2 obtained after a single-step reduction of S_1 is also good. As an example we argue the case for the identifier lookup axiom shown below:

$$\frac{Scope(H, L, \text{“}x\text{”}) = ln}{H, L, x \xrightarrow{e} H, L, ln\text{*“}x\text{”}}$$

If the state H, L, x is good then x must be named in the set $\mathcal{G} \setminus \mathcal{B}$. As a result the state H, L, ln*“x” obtained after reduction would also be good. The only property accessed during this reduction would be “x” which is the set $\mathcal{N}$(“x”). Therefore by definition $Safe^h(H, L, x)$ holds. This completes the argument for the identifier lookup axiom.

Inductive case. The inductive case is the set of contextual rules. The general form of a contextual rule for a context C is

$$\frac{S_1 \rightarrow S_2}{C[S_1] \rightarrow C[S_2]}$$

We show that all good states $HC[s_1]$ can be reduced to a good state of the form $C[s_2]$ o in one or more steps, such that, all states appearing on $subTr(C[S_1], C[S_2])$ are safe.

For all contexts C except va[_], the argument goes as follows. From Proposition 2 it follows that if a state $C[S_1]$ is good then the state S_1, assuming the context C is different from va[_], is also good. From the induction hypothesis, there exists a good state S_2 such that $S_1 \rightsquigarrow S_2$ and $Safe^h(subTr(S_1, S_2))$ are safe. By the contextual rule, it follows that $C[S_1] \rightsquigarrow C[S_2]$. and all states appearing on the reduction sub-trace from $C[S_1]$ to $C[S_2]$ are of the form $C[S]$ where S is a state appearing on the reduction sub-trace from S_1 to S_2. Since the state S_2 is good, it follows from Proposition 2 (part 2) that the state $C[S_2]$ is also good. Since $Safe^h(subTr(S_1, S_2))$ holds, it follows from Proposition 1 that $Safe^h(subTr(C[S_1], C[S_2]))$ also holds. This

completes the argument.

For the context $C_{index} := va[\_]$, we argue as follows. For all good states S_1 that can be reduced using the contextual rule for C_{index}, $term(S_1)$ is of the form $va[e]$, where the expression e is of one of the following forms: (A) expression IDX(e') for some expression e' such that $TGood^h(e')$. (B) String m such that $m \notin \mathcal{B}$. For case (A) by symbolically execution and show that the state S_1 can be reduced in many steps to a good state S_2 such that $term(S_2)$ is of the form $l[m]$ where $m \notin \mathcal{B}$ and $Safe^h(S_1, S_2)$. We use properties from Proposition 3 to symbolically execute through function invocations. For case (B), a transition axiom applies to the state S_1 and proof follows from the hypothesis. This completes the argument for the inductive case. □

Restatement of Theorem 3. *For all global variable whitelists $\mathcal{G}$ such that $\mathcal{P}_{nat} \subseteq \mathcal{G}$, for all third-party terms $t \in Terms^{ES3}_{user}$, the term $\phi^h(t)$ satisfies the isolation property $Isolation^h_{\mathcal{G}}(H^h, L^h, \phi^h(t))$.*

Proof Sketch: Follows from Proposition 1 and Lemmas 2, 3 and 4. □

A.3 Proofs from Chapter 6

In this section, we prove theorems 11, 6 and 7 from Chapter 6. All proofs are carried out for the language $ES3$ augmented with the freezeAll statement. The map $Auth^m$ is defined in Section 6.3 of Chapter 6.

A.3.1 Preliminaries

Theorem 11 states authority isolation implies isolation. We prove this theorem by an induction on the number of components. Theorem 7 states that the initial authorities of mashup components components sandboxed under the mechanism $\mathcal{S}^m_{\mathcal{G}}$ are isolated. The proof of this theorem is straightforward and involves showing that the authorities of the var ious sandboxed components under the map $Auth^m$ are isolated. Theorem 6 states that $Auth^m$ is a safe authority map. This is the most tricky theorem to

prove. We set up the following notations and definitions to prove this theorem. We split Theorem 6 into three lemmas that prove: *Sufficiency* (Lemma 7), *Only Connectivity begets Connectivity* (Lemma 8) and *No Authority Amplification* (Lemma 9) respectively for the map $Auth^m$.

Sufficiency. We begin by setting up some notation and definitions. A state S is called a user state iff $term(S) \in Terms^{ES3}_{user}$.

Definition 25 (Sufficiency at a state) *An authority map Auth is sufficient at a state S, denoted by Suff*$(Auth, S)$*, iff*

$$Act(\tau(S)) \cap Actions(heap(S)) \subseteq Auth(S).$$

An authority map $Auth$ is sufficient iff it is sufficient at all user states.

Definition 26 (State safety) *A state S is safe with respect to an authority map Auth and an initial state* S_0 *iff* $S_0 \rightsquigarrow S$ *and* $Act(S) \cap Actions(heap(S_0)) \subseteq Auth(S_0)$*. This property is denoted by* $Safe^m(Auth, S_0, S)$*.*

The property $Safe^m(Auth, S_0, S)$ can be extended to traces τ by defining $Safe^m(Auth, S_0, \tau)$ as true iff $\tau[0] = S_0$ and for all $S \in \tau$, $Safe^m(Auth, S_0, S)$ holds.

Proposition 4 *For all authority maps Auth and all user states S,*

$$Safe^m(Auth, S, \tau(S)) \implies Suff(Auth, S).$$

Proof Sketch: Follows from the definitions of the $Safe^m$ and *Suff* and the fact that $Act(\tau(S)) := \bigcup_{S' \in \tau(S)} Act(S')$. □

In light of the above proposition, we can prove that the map $Auth^m$ is sufficient at a state S by showing that the safety property $Safe^m(Auth^m, S, \tau(S))$ holds for the S. In what follows, we describe our technique for proving sufficiency for the map $Auth^m$.

By definition of the map $Auth^m$, we have that $Auth^m$ is trivially sufficient at all user states that are not $\mathcal{S}^m_{\mathcal{G}}$-consistent. By definition of $\mathcal{S}^m_{\mathcal{G}}$-consistency, showing sufficiency for all $\mathcal{S}^m_{\mathcal{G}}$-consistent states is equivalent to showing sufficiency for all states in the set $Init^m := \{H^m_{\mathcal{G}}, L^m_{\mathcal{G}}, \phi^h(t) \mid t \in Terms^{ES3}_{user}\}$. We argue sufficiency for such traces using a technique very similar to the one used in Section A.2 for arguing safety

for all traces beginning with states in $Init^h$. We define a property $Good^m(S_0, S)$ for all initial $\mathcal{S}^m_{\mathcal{G}}$-consistent state S_0 and intermediate 3state S and show the following: (1) (*Initial Goodness*) $Good^m(S_0, S_0)$ holds for all states $S_0 \in Init^m$ (2) (*Preservation*) For all non-final state S such that $Good^m(S_0, S)$ holds, there exists a state S_2 such that $S_1 \rightarrow *S_2$, $Good^m(S_0, S_2)$ holds, and $Safe^m(Auth^m, S_0, S)$ holds for all states S in the sub-trace $subTr(S_1, S_2)$.

The predicate $Good^m(S_0, S)$ is defined using the predicate $Good^h(S)$ defined in Section A.2, and the following notations. We use $RLocs(S_0)$ for the set $\{l \mid \forall p : p \in Props \setminus \mathcal{B} \implies l, p, \mathsf{r} \in Auth^m(S_0)\}$, and $L^m_{wrapped}$ for the set of locations of all the wrapper function objects created by the initialization code T_{wrap}. For an object at location $l \in L^m_{wrapped}$, the function expression stored in the `@body` property is denoted by let $expr(l)$, and has the form:

```
function(){var $this = (this==({ }).$g? { }: $this); return ({ }).$e_x.call($this)}
```

where e_x is a string unique to the expression.

Definition 27 (Goodness $Good^m$) *$Good^m(S_0, S)$ holds for states S_0, S iff the following properties hold:*

(1) *$Good^h(S)$ holds.*

(2) *$S_0 \rightsquigarrow S$.*

(3) *Structure of $term(S)$ does not contain any heap locations from L^m.*

(4) *All identifiers present in $term(S)$ must be contained in the set $\mathcal{N}(term(S_0))$.*

(5) *For all functions objects stored on the heap at locations outside the set $L^m_{wrapped}$, must have function bodies with all identifiers in the set $\mathcal{N}(term(S_0))$.*

(6) *For all heap locations l present in $stack(S)$ or $term(S)$, either $l = \#global$ or $l \in RLocs(S_0)$.*

(7) *For all $l \in L^m_{wrapped}$, $heap(S)(l).$`@body` $= expr(l)$.*

(8) *For all edges* l_1, p, l_2 *in graph* $Gr(heap(S))$ *edge* l_1, p, l_2 *in graph* $Gr(heap(S_0))$ *OR the following holds:*

(a) $l_1 \in dom(heap(S_0)) \implies l_1 \in RLocs(S_0)$

(b) $l_2 \in dom(heap(S_0)) \implies l_2 \in RLocs(S_0)$

Before describing the main results we describe a proposition on good states. Given a heap H and set of location $\mathcal{L}$, let $\mathcal{FN}(H, \mathcal{L})$ denote the union of the set of identifier names associated with function objects referenced by locations in $\mathcal{L}$. Formally, $\mathcal{FN}(H, \mathcal{L}) := \{\mathcal{N}(H(l).\texttt{@body}) \mid l \in Funcs_H(\mathcal{L})\}$. For any location l such that $l \neq \#global$, let $LAuth(H, L, l)$ be the authority associated with the location, defined as the set $(\mathcal{L} \cup IdAuth_H(H, L, \mathcal{FN}(H, \mathcal{L}), \emptyset)) \setminus A_{\not w}$ where $\mathcal{L} := Reach(Gr^H, l, Props \setminus \mathcal{B})$ and $A_{\not w} := \big(dom(H^m_{\mathcal{G}}) \setminus \{\#global\}\big) \times Props \times \{\mathsf{w}\}$.

Proposition 5 *Given a* $\mathcal{S}^m_{\mathcal{G}}$*-consistent state* $S_0 := H_0, L_0, t_0$ *and a state* $S := H, L, t$*, if* $Good^m(S_0, S)$ *holds then for all locations* l *in* $RLocs(S_0) \cup dom(H) \setminus dom(H_0)$*,* $LAuth(H, L, l) \subseteq Auth^m(S_0)$*.*

Proof Sketch: By definition,
$LAuth(H, L, l) = (\mathcal{L} \cup IdAuth_H(H, L, \mathcal{FN}(H, \mathcal{L}), \emptyset)) \setminus A_{\not w}$, where $\mathcal{L}$ is the set $Reach(Gr^H, l, Props \setminus \mathcal{B})$ and $A_{\not w}$ is the set $\big(dom(H^m_{\mathcal{G}}) \setminus \{\#global\}\big) \times Props \times \{\mathsf{w}\}$. Using the definition of $Good^m(S_0, S))$, it is easy to show that $\mathcal{L} \setminus A_{\not w}$ is a subset of $Auth^m(S_0)$. Furthermore, from $Good^m(S_0, S))$ it follows that $\mathcal{FN}(H, \mathcal{L}) \subseteq \mathcal{N}(term(S_0))$. We show by induction that for all property names P such that $P \subseteq \mathcal{N}(term(S_0))$, $IdAuth_H(H, L, P, \emptyset) \subseteq Auth^m(S_0)$. This completes the proof. □

Only Connectivity begets Connectivity. We first prove a proposition that (Proposition 6) that for any two heaps H and K, if the $Auth^m(K, L, t)$ contains an action a that is not present in the $Auth^m(H, L, t)$ then there must be a common read action l, p, r in both $Auth^m(K, L, t)$ and $Auth^m(H, L, t)$ such that H and K do not agree on the location-property l, p and action a is presentin $LAuth(H, L, a)$. This proposition lets us prove the only connectivity begets property by contradiction.

No Authority Amplification. The property holds trivially during the execution of states that are not $\mathcal{S}^m_{\mathcal{G}}$-consistent. We argue the property for all $\mathcal{S}^m_{\mathcal{G}}$-consistent states by contradiction. The proof makes use of the proposition described in the above paragraph and the goodness property, provided by Lemmas 5 and 6, which states that for all $\mathcal{S}^m_{\mathcal{G}}$-consistent states S_0, if the execution of S_f terminates in a final state S_f then $Good^m(S_0, S_f)$ holds.

A.3.2 Main Results

Theorem 11 *For all basic mashups $m = Mashup((t_1, id_1), \ldots, (t_n, id_n))$, all global variable whitelists $\mathcal{G}$, heaps H, stacks L, such that $Wf(H, L, prog(m))$:*

$$AuthIsolation(H, L, m, \mathcal{G}) \implies Isolation^m_{\mathcal{G}}(H, L, m, \mathcal{G}).$$

Proof Sketch: Consider a mashup $m = Mashup((t_1, id_1), \ldots, (t_n, id_n))$, a heap H and stack L such that $H, L, prog(m)$ is well formed. Furthermore, let $\mathcal{G}$ be a global variable whitelist such that $AuthIsolation(H, L, m, \mathcal{G})$ holds. From the definition of *AuthIsolation* it follows that there exists a safe authority map *Auth* satisfying the following conditions.

$$\forall i, j : i < j \Rightarrow Auth(H, L, t_i) \not\triangleright Auth(H, L, t_j) \tag{A.5}$$

$$\forall i : \forall a : (a \in Auth(H, L, t_i) \ \wedge \ loc(a) = \#global) \implies props(a) \in \mathcal{G} \tag{A.6}$$

In order to prove this theorem we first show that $AuthIsolation(H, L, m, \mathcal{G})$, implies the following property

$$\forall i, j : 1 \leq i \leq j \leq \kappa \implies Auth(H_i, L_i, t_j) = Auth(H, L, t_j). \tag{A.7}$$

where $\kappa = Abnormal(H, L, prog(m))$ and for all $i \leq \kappa$, $H_i, L_i = HS(H, L, \{t_1 \ldots t_n\}, i)$. For conciseness, in the rest of the proof we use κ to denote $Abnormal(H, L, prog(m))$. We prove this property by induction over i. The base case ($i = 1$) is trivial. Assume the property holds for $i = k$. Therefore,

$$\forall j : k \leq j \implies Auth(H_k, L_k, t_j) = Auth(H, L, t_j) \tag{A.8}$$

In order to prove the inductive case ($i = k+1$) we consider any $j \geq k+1$. From Condition A.8 it follows that $Auth(H_k, L_k, t_j) = Auth(H, L, t_j)$ and $Auth(H_k, L_k, t_k) = Auth(H, L, t_k)$. Combining with Condition A.5 it follows that

$$Auth(H_k, L_k, t_k) \not\triangleright Auth(H_k, L_k, t_j) \tag{A.9}$$

From the definition of the map HS, we have that there exists a value co such that $H_k, L_k, t_k \rightsquigarrow H_{k+1}, L_{k+1}, co$. Since $Auth$ is a safe authority, from Condition A.9 and the "only connectivity begets connectivity" property for $Auth$, it follows that $Auth(H_{k+1}, L_{k+1}, t_j) = Auth(H_k, L_k, t_j) = Auth(H, L, t_j)$ (the last equality follows from Condition A.8). Thus the inductive case holds and $AuthIsolation(H, L, m, \mathcal{G})$ implies Condition A.7. We now show that $AuthIsolation(H, L, m, \mathcal{G})$ and Condition A.7 imply $Isolation^m_{\mathcal{G}}(H, L, m, \mathcal{G})$.

In order to prove $Isolation^m_{\mathcal{G}}(H, L, m, \mathcal{G})$, it is sufficient show that for all i, j such that $1 \leq i < j \leq \kappa$:

$$Act(\tau(H_i, L_i, t_i)) \not\triangleright Act(\tau(H_j, L_j, t_j)) \tag{A.10}$$

$$\forall a : (a \in Act(\tau(H_i, L_i, t_i)) \ \wedge \ loc(a) = \#global) \implies props(a) \in \mathcal{G} \tag{A.11}$$

Since $Auth$ is a safe authority map, using Condition A.7 and the "sufficiency" property for $Auth$, we have:

$$\forall i : 1 \leq i \leq \kappa \implies Act(\tau(H_i, L_i, t_i)) \ \cap \ Actions(H_i) \subseteq Auth(H, L, t_i) \tag{A.12}$$

From the semantics of $ES3$, it follows that:

$$\forall i : 1 \leq i < \kappa : \left(\begin{array}{l} Act(\tau(H_i, L_i, t_i)) \subseteq Actions(H_{i+1}) \\ \wedge \ Actions(H_i) \subseteq Actions(H_{i+1}) \end{array} \right) \tag{A.13}$$

Condition A.10 directly follows from Condition A.5, Condition A.12, and Condition A.13. Since the global object $\#global \in dom(H) \subseteq dom(H_i)$, Condition A.11 directly follows from Condition A.12 and Condition A.6. □

We now prove that the sandboxing mechanism $\mathcal{S}^m_{\mathcal{G}}$ is well formed.

Restatement of Theorem 5. *$\mathcal{S}^m_{\mathcal{G}}$ is a well-formed mashup sandboxing mechanism*

Proof Sketch: For $i \in \{1, \ldots, n\}$, let $\phi_i := \alpha_i \circ \phi^h$. To prove this theorem, we show that $Wf_h(H^m_{\mathcal{G}}) \wedge Wf_s(L^m_{\mathcal{G}}, H^m_{\mathcal{G}})$ hold and for all terms $t \in Terms^{ES3}_{user}$, for each $i \in \{1, \ldots, n\}$, if $Wf_t(t, H)$ holds then $Wf_t(\phi_i(t), H)$ holds. H^h, L^h are defined as the final heap and stack obtained on executing the term $T_{wrap}; T_{freeze}$. Since this term is well formed, from Theorem 1 (preservation part) it follows that $H^m_{\mathcal{G}}, L^m_{\mathcal{G}}$ are well formed. Therefore $Wf_h(H^h) \wedge Wf_s(L^h, H^h)$ holds. From the definition of α_i and ϕ^h (see Definition 4) and the grammar for user-level ES3 terms (see figures 4.2 and 4.3) it follows that, if $t \in Terms^{ES3}_{user}$ then $\phi_i(t) \in Terms^{ES3}_{user}$. As a consequence, if $Wf_t(t, H^m_{\mathcal{G}})$ holds then $Wf_t(\phi_i(t), H^m_{\mathcal{G}})$ holds. □

Next, we move to proving that the map $Auth^m$ is safe.

Lemma 5 *(*Init*) $Good^m(S_0, S_0)$ holds for all $\mathcal{S}^m_{\mathcal{G}}$-consistent states S_0. for any black-list $\mathcal{B}$, for all programs in $P \in J_{sub}(\mathcal{B})$ with program id pid.*

Proof Sketch: Consider an $\mathcal{S}^m_{\mathcal{G}}$-consistent state S_0. Therefore it follows that $term(S_0) \in Terms^{ES3}_{user}$. Properties $2-8$ of the predicate $Good^m$ hold trivially. By definition $term(S_0) \in Terms^{ES3}_{user}$ and there exists a state S' such that $term(S')$ is of the form $\phi^h(t')$ for some $t' \in Terms^{ES3}_{user}$, such that $S' \rightsquigarrow S_0$. Similar to proof of Lemma 3, we can prove that $Good^h(S')$ holds. From Lemma 4, it follows that $Good^h(S_0)$ holds and thus property 1 of the predicate $Good^m$ holds as well. □

Lemma 6 *For any $\mathcal{S}^m_{\mathcal{G}}$-consistent state S_0, and for any non-final state S_1 such that $Good^m(S_0, S_1)$ holds, there exists a S_2 such that $S_1 \rightsquigarrow S_2$, $Good^m(S_0, S_2)$ holds and $Safe^m(Auth^m, S_0, S)$ holds for all states S in the sub-trace $subTr(S_1, S_2)$.*

Proof Sketch: Consider a $\mathcal{S}^m_{\mathcal{G}}$-consistent state S_0, and a non-final state S_1 such that $Good^m(S_0, S_1)$ holds. We express $Good^m(S_0, S_1)$ as $Good^h(S_1) \wedge Good^m_{rest}(S_0, S_1)$ where $Good^m_{rest}(S_0, S_1)$ is the conjunction of properties 2-8 from definition 27. We prove the Lemma by showing the following properties

A. If $Good^h(S_1)$ holds then there exists a state S_2 such that $S_1 \rightsquigarrow S_2$, $Good^h(S_2)$ holds and $Safe^h(S)$ holds for all states S in $subTr(S_1, S_2)$

B. If $Good^m_{rest}(S_0, S_1)$ holds and $Safe^h(S_1)$ holds then for all states S_2 such that $S_1 \rightarrow S_2$, $Good^m_{rest}(S_0, S_2)$ holds.

Property A holds from Lemma 4 for all $ES3$ states except those that involve the term freezeAll. For the term freezeAll we can argue by symbolic execution (similar to the proof of Proposition prop:hpimanual). Property B is proved by a straightforward induction on the set of reduction rules. The transition axioms form the base cases and the contextual rules form the inductive cases. □

Lemma 7 *[Sufficiency] For all well-formed states S, $Act(\tau(S)) \subseteq Auth^m(S)$.*

Proof Sketch: Follows from Proposition 4, and Lemmas 5 and 6. □

Proposition 6 *For all terms t, stacks L and states H, K: IF there exists an action a such that $a \in Auth^m(K, L, t)$ and $a \notin Auth^m(H, L, t)$ THEN there exists an action $l, p, \mathsf{r} \in Auth^m(K, L, t) \cap Auth^m(H, L, t)$ such that $H(l).p \neq K(l).p$ and $a \in LAuth(K, L, l)$.*

Proof Sketch: For all states K, L, t $Auth^m(K, L, t)$ consists of a state-independent constant portion $A_1 := \{\#global\} \times (\mathcal{P}_{nat} \cup Props_{int}) \times \{\mathsf{r}\}$ and a state-dependent portion $IdAuth_H(K, L, \mathcal{N}(t), \emptyset)$. It is sufficient to show that: IF there exists an action a such that $a \in IdAuth_H(K, L, \mathcal{N}(t), \emptyset)$ and $a \notin IdAuth_H(H, L, \mathcal{N}(t), \emptyset)$ THEN there exists an action $l, p, \mathsf{r} \in IdAuth_H(K, L, \mathcal{N}(t), \emptyset) \cap IdAuth_H(H, L, \mathcal{N}(t), \emptyset)$ such that $H(l).p \neq K(l).p$ and $a \in LAuth(K, L, l)$. We prove this property by induction over the recursion length of the function $IdAuth_H$. □

Lemma 8 *[Only Connectivity begets Connectivity] For all well-formed states H, L, t such that $t \in Terms^{ES3}_{user}$, and $\tau(H, L, t)$ terminates in a final state H_f, L_f, co_f, the following holds: For all terms $u \in Terms^{ES3}_{user}$, $Act(\tau(H, L, t)) \not\bowtie Auth^m(H, L, u)$ implies $Auth^m(H, L, u) = Auth^m(H_f, L_f, u)$.*

Proof Sketch: Consider a well-formed state $S := H, L, t$ such that $S_f := H_f, L_f, t$ terminates in a final state S_f. From the semantics of ES3 we have that $L = L_f$. We prove the lemma by contradiction.

Suppose the lemma does not hold. Then there exists a term $u \in Terms^{ES3}_{user}$ such that $Act(\tau(H,L,t)) \not\triangleright Auth^m(H,L,u)$ and $Auth^m(H,L,u) \neq Auth^m(H_f,L_f,u)$. From the latter and Proposition 6, we have that there exists an action $l,p,\mathsf{r} \in Auth^m(H,L,u)$ such that $H(l).p \neq H_f(l).p$. Since the initial and final heaps differ on property p of location l it must be the case that this property was written during the reduction of state H,L,t. Therefore $l,p,\mathsf{w} \in Act(\tau(H,L,t)$. Since $l,p,\mathsf{r} \in Auth^m(H,L,u)$, $act(\tau(H,L,t)) \triangleright Auth^m(H,L,u)$. Thus our assumption was wrong and the lemma is true. □

Lemma 9 *[No Authority Amplification] For all well-formed states H,L,t such that $t \in Terms^{ES3}_{user}$, and $\tau(H,L,t)$ terminates in a final state H_f,L_f,co_f, the following holds: For all terms $u \in Terms^{ES3}_{user}$, $acc(\tau(H,L,t)) \triangleright Auth^m(H,L,u)$ implies*

$$Auth^m(H_f,L_f,u) \subseteq \left(\begin{array}{l} Auth^m(H,L,u) \cup Auth^m(H,L,t) \\ \cup\ Actions(H_f) \setminus Actions(H) \end{array} \right).$$

Proof Sketch: Consider a well-formed state $S := H,L,t$ such that $S_f := H_f,L_f,t$ terminates in a final state S_f. From the semantics of ES3 we have that $L = L_f$. If S is not $\mathcal{S}^m_{\mathcal{G}}$-consistent then $Auth^m(S) = Actions(heap(S))$. Therefore the lemma holds trivially for such states. If S is $\mathcal{S}^m_{\mathcal{G}}$-consistent then we first note that from Lemmas 5 and 6, $Good^m(S,S_f)$ holds. We now prove the lemma by contradiction.

Suppose the lemma does not hold. Then there exists a term $u \in Terms^{ES3}_{user}$ such that $Act(\tau(H,L,t)) \triangleright Auth^m(H,L,u)$ and
$Auth^m(H,L,u) \cap Actions(H) \not\subseteq Auth^m(H,L,u) \cup Auth^m(H,L,t)$. The latter implies that there exists an action $a \in Auth^m(H_f,L,u) \cap Actions(H)$ such that $a \notin Auth^m(H,L,u)$ and $a \notin Auth^m(H,L,t)$. From Proposition 6 we have that there exists an action l,p,r in $Auth^m(H,L,u) \cap Auth^m(H_f,L,u)$ such that $H(l).p \neq H_f(l).p$ and $a \in LAuth(H_f,L,l)$. It follows that $l \in dom(H)$. Since $Good^m(S,S_f)$ holds, from Proposition 5 we have that $LAuth(H_f,L,l) \subseteq Auth^m(H,L,t)$. Therefore $a \in LAuth(H,L,t)$. This means that our assumption was wrong and the lemma holds. □

Restatement of theorem 6. *$Auth^m$ is a safe authority map for the language ES3 augmented with the freezeAll statement.*

Proof Sketch: Follows from Lemmas 7, 8 and 9. □

Restatement of Theorem 7. *For all basic mashups m and for all $\mathcal{P}_{nat}$-compatible and id-compatible whitelists $\mathcal{G}$, the mashup $\mathcal{S}^m_{\mathcal{G}}\langle m\rangle$ achieves authority isolation for the language ES3 augmented with the* freezeAll *statement.*

Proof Sketch: Let $(s_1, id_1), \ldots, (s_n, id_n)$ be the (sandboxed) components of the mashup $\mathcal{S}^m_{\mathcal{G}}\langle m\rangle$. We prove that authority isolation holds by showing that the following properties hold for the authority map $Auth^m$:

A. $\forall i, j : i < j \Rightarrow Auth^m(H^m_{\mathcal{G}}, L^m_{\mathcal{G}}, s_i) \not\triangleright Auth^m(H^m_{\mathcal{G}}, L^m_{\mathcal{G}}, s_j)$

B. $\forall i : \forall a : (a \in Auth^m(H^m_{\mathcal{G}}, L^m_{\mathcal{G}}, s_i) \ \wedge \ loc(a) = \#global) \implies props(a) \in \mathcal{G}$

By definition of $Auth^m$, for all terms s, all actions $a \in Auth^m(H^m_{\mathcal{G}}, L^m_{\mathcal{G}}, s)$ satisfy the following: (1) if $loc(a) \in dom(H^m_{\mathcal{G}}) \setminus \{\#global\}$ then $perm(a) = \mathsf{r}$. (2) if $loc(a) = \#global$ then $perm(a) = \mathsf{r}$ iff $props(a) \in \mathcal{P}_{nat} \cup Props_{int}$, (3)) if $loc(a) = \#global$ then $perm = \mathsf{w}$ iff $props(a) \in \mathcal{N}(s)$.
Property A. From the definition of the mechanism $\mathcal{S}^m_{\mathcal{G}}$ we have that for any two sandboxed components (s_i, id_i) and (s_j, id_j), $\mathcal{N}(s_i) \cap \mathcal{N}(s_j) = \emptyset$. Property A. immediately follows from this fact and properties (1), (2) and (3).
Property B. Since $\mathcal{G}$ is $\mathcal{P}_{nat}$-compatible and id-compatible, $\mathcal{P}_{nat} \subseteq \mathcal{G}$ and for all components (s_i, id_i), $\mathcal{N}(s_i) \subseteq \mathcal{G}$ (since $\mathcal{S}^m_{\mathcal{G}}$ ensures that all sandboxed terms s_i have all identifiers prefixed with id_i. Property B. immediately follows from this fact and properties (2) and (3). □

A.4 Proofs from Chapter 7

In this section we prove Theorems 8 and 9 from Chapter 7 using the SES-light analysis framework developed in Section 7.3.

A.4.1 Preliminaries

Similar to Theorem 1, the progress part of Theorem 8 is proven using an induction on terms and a the preservation part by an induction on the reduction rules. Both the inductions are straightforward and therefore we describe the proof briefly.

In order to prove Theorem 9 we show by induction on the set of reduction rues that α-renaming of states is preserved under reduction. We first precisely define the map $Rn(S, \alpha)$ for a given a program state S and variable renaming map α.

State renaming. Renaming of states is defined by individually renaming the heap, stack and term parts. As discussed in Section 7.3.4, the renaming of terms is based on the label of the closest binding-scope. We formalize this by defining a renaming map $Rn_s(s, \alpha, \kappa)$ that renames a statement s using the variable renaming map α and an explicit binding-scope map $\kappa : \mathit{Vars} \rightharpoonup \mathcal{L}$ that maps variable names to labels of the scope in which they are bound. The initial binding-scope map for renaming the term is obtained from the current stack. For any given stack $A := ERG{:}R_1{:}\ldots{:}R_n$, we define the scope-$\kappa_A$ as the map $\kappa_{R_1}{:}\ldots{:}\kappa_{R_n}$. Here κ_R denotes the scope-binding map $dom(R) \rightarrow Lab(R)$ that maps all variables in $dom(R)$ to $Lab(R)$, and the notation $\kappa_1{:}\kappa_2$ for any two scope-binding maps κ_1, κ_2 denotes a map that behaves as κ_2 on all variables in $dom(\kappa_2)$, as κ_1 on all variables $x \in dom(\kappa_1) \setminus dom(\kappa_2)$, and is undefined on all other variables.

The formal definition for Rn_s for all statements is provided in Figures A.1 and A.2. The key idea is to recursively rename all top-level identifiers using the map α and with respect to label provided by the map κ. We update the map κ across recursive calls using the map $k(s, \kappa)$ which is also defined alongside Rn_s in Figures A.1 and A.2.

We define renaming for activation records, stacks, closures and heaps as follows. An activation record R is renamed by replacing each variable $x \in dom(R)$ with $\alpha(x, Lab(R))$. A stack A is renamed by renaming all activation records appearing on it. A closure (s, A) is renamed by individually renaming the stack A with respect to α, and renaming the term s with respect to α and κ_A. Finally a heap H is renamed by renaming all closures appearing on it.

Definition 28 *Given a program state $S := H, A, t$, the renamed state $Rn(S, \alpha)$ is defined as $Rn(H, \alpha), Rn(A, \alpha), Rn_s(t, \alpha, \kappa_A)$ where $Rn(H, \alpha)$ and $Rn(A, \alpha)$ are renamed heap H and stack A respectively.*

A.4.2 Main Results

Restatement of Theorem 8. *For all states S_1 such that $Wf(S_1)$ holds:*

- *(Preservation) If there exists a state S_2 such that $S_1 \rightarrow S_2$, then $Wf(S_2)$ holds.*
- *(Progress) If $term(S_1) \notin \{N\} \cup \{Th(v) \mid v \in Vals\}$ then there exists a state S_2 such that $S_1 \rightarrow S_2$.*

Proof Sketch: The proof of the above theorem is very similar to the proof of Theorem 1. The preservation part is proven by an induction on the set of reduction rules. The transition axioms form the base case and the contextual rules form the inductive case. As an example we give the proof for the assignment axiom

$$\frac{CanPutVar(A, y) \qquad K, B = Update(H, A, y, v, \{\mathsf{writable}\})}{H, A, y{=}v \rightarrow K, B, \mathsf{N}}$$

If the state $H, A, y{=}v$ is well formed then the heap H and term $y{=}v$ are well formed. From this and the definition of *Update* we can show that the heap K is also well formed. As a result the state K, B, N is well formed and thus well-formedness is preserved.

The progress part is proven by a structural induction on the set of terms. As an example consider the expression $y{=}e$. If e is not a value then the contextual rule for evaluating expressions applies for the context $y{=}\_$. On the other hand if e is a value v then the transition axiom for the assignment expression $y{=}v$ applies. □

We now move to proving Theorem 9. The proof is based on a key bisimulation lemma (Lemma 10) that says that for variable-renaming maps α, a state S_1 reduces to S_2 iff the state $Rn(S_1, \alpha)$ reduces to $Rn(S_2, \alpha)$.

Proposition 7 *For all states $S_1 := H_1, A_1, t_1$ and $S_2 := H_2, A_2, t_2$ such that $S_1 \rightarrow S_2$ then $k(t_1, \kappa_{A_1}) = k(t_2, \kappa_{A_2})$.*

$Rn_s : Stmts \times (Vars \times \mathcal{L} \to Vars) \times (Vars \rightharpoonup \mathcal{L}) \to Terms$

$k : Stmts \times (Vars \rightharpoonup \mathcal{L}) \to (Vars \rightharpoonup \mathcal{L})$

Let $\alpha_\kappa(e)$ be $\alpha(e, BV(e))$ if $BV(e)$ is defined, and e otherwise.

s	$Rn_{\mathbf{s}}(\mathbf{s}, \alpha, \kappa)$	$k(s, \kappa)$
y= e	$\alpha_\kappa(y)$= $\alpha_\kappa(e)$	κ
y= e_1 BIN e_2	$\alpha_\kappa(y)$= $\alpha_\kappa(e_1)$ BIN $\alpha_\kappa(e_2)$	κ
y= UN e	$\alpha_\kappa(y)$= UN $\alpha_\kappa(e)$	κ
y= e_1[e_2,a]	$\alpha_\kappa(y)$= $\alpha_\kappa(e_1)$[$\alpha_\kappa(e_2)$,a]	κ
e_1[e_2,an] = e_3	$\alpha_\kappa(e_1)$[$\alpha_\kappa(e_2)$,an] = $\alpha_\kappa(e_3)$	κ
y= {x : e}	$\alpha_\kappa(y)$= {x : ($\tilde{\alpha_\kappa}(e)$)}	κ
y= [$\tilde{e}$]	$\alpha_\kappa(y)$= [$\tilde{\alpha_\kappa}(e)$]	κ
y= e($\tilde{e_i}$)	$\alpha_\kappa(y)$= $\alpha_\kappa(e)$($\tilde{\alpha_\kappa}(e_i)$)	κ
y= e[e',an]($\tilde{e_i}$)	$\alpha_\kappa(y)$= $\alpha_\kappa(e)$[$\alpha_\kappa(e')$,an]($\tilde{\alpha_\kappa}(e_i)$)	κ
y= new e($\tilde{e_i}$)	$\alpha_\kappa(y)$= new $\alpha_\kappa(e)$($\tilde{\alpha_\kappa}(e_i)$)	κ
eval$_{\text{nf}}$(e)	eval$_{\text{nf}}$($\alpha_\kappa(e)$)	κ
return e	return $\alpha_\kappa(e)$	κ
var x	var $\alpha_\kappa(e)$	κ
throw e	throw $\alpha_\kappa(e)$	κ
y= function x($\tilde{z}$){s_1}	$\alpha_\kappa(y)$= function $\alpha_\kappa(x)$ ($\tilde{z}$){$Rn(s_1, \alpha, \kappa_r)$} where $\kappa_r := \kappa$:{$\tilde{z}$} $\to Lab(s)$:$BV(s_1) \to Lab(s)$)	κ
function x($\tilde{z}$){s_1}	function $\alpha_\kappa(x)$ ($\tilde{z}$){$rn(s_1, \alpha, \kappa_r)$} where $\kappa_r := \kappa$:{$\tilde{z}$} $\to Lab(s)$:$BV(s_1) \to Lab(s)$)	κ

Figure A.1: Statement renaming in SES-light (Part 1)

$\mathbf{s}$	$Rn_{\mathbf{s}}(\mathbf{s}, \alpha, \kappa)$	$k(s, \kappa)$
$s_1;s_2$	$Rn_s(s_1, \alpha, \kappa);Rn_s(s_2, \alpha, \kappa_1)$ where $\kappa_1 = k(s_1, \kappa)$	$k(s_2, \kappa_1)$
if (e) then s_1 else s_2	if $(\alpha_\kappa(e))$ then $Rn_s(s_1, \alpha, \kappa)$ else $Rn_s(s_2, \alpha)$	κ
while (e) s_1	while $(\alpha_\kappa(e))$ $Rn_s(s_1, \alpha, \kappa)$	κ
for $(x$ in $e)$ s_1	for $(x$ in $\alpha_\kappa(e))$ $Rn_s(s_1, \alpha, \kappa)$	κ
try$\{s_1\}$catch$(x)\{s_2\}$ finally$\{s_3\}$	try$\{Rn_s(s_1, \alpha, \kappa)\}$ catch$(x)\{Rn(s_2, \alpha, \kappa_c)\}$ finally$\{Rn_s(s_3, \alpha, \kappa_f)\}$ where $\kappa_c := k(s_2, \kappa)$:$\{x\} \to Lab(s))$ $\kappa_f := k(s_3, \kappa_c)$	$k(s_2, \kappa_f)$
@TS(x,v)	@TS($\alpha_\kappa(x)$,v)	κ
@TS-help(x,e)	@TS-help($\alpha_\kappa(x)$, α_κ(e))	κ
@TN(x v)	@TS($\alpha_\kappa(x)$,v)	κ
@TN-help(x, e)	@TS($\alpha_\kappa(x)$,α_κ(e))	κ
@TB(x,v)	@TB($\alpha_\kappa(x)$,v)	κ
@TO(x,v)	@TO($\alpha_\kappa(x)$,v)	κ
@TP(x, v, pn)	@TP($\alpha_\kappa(x)$,v,pn)	κ
@TP-help(x, e, v, pn)	@TP-Help($\alpha_\kappa(x)$,α_κ(e), v, pn)	κ
@TP-check(x, e)	@TS($\alpha_\kappa(x)$,α_κ(e))	κ
@Fun1(tag, A, x, e_f, e_t, $\tilde{e}$)	@Fun1(tag, A, $\alpha_\kappa(x)$, $\alpha_\kappa(e_f)$, $\alpha_\kappa(e_t)$, $\alpha_\kappa\tilde{(e)}$)	κ
@Fun2(tag, A, x, v, fv)	@Fun2(tag, A, $\alpha_\kappa(x)$, v, $Rn_s(fv, \alpha, \kappa)$)	κ_A
@Fun3(tag, A, x, v, s)	@Fun3(tag, A, $\alpha_\kappa(x)$, v, $Rn_s(s, \alpha, \kappa)$	κ_A
@Catch1(x, v, s)	@Catch1($\alpha_\kappa(x)$, v, $Rn_s(s, \alpha, \kappa$:$\{x\} \to Lab(s)))$	κ
@EVAl(A, s)	@Eval(A, $Rn_s(s, \alpha, \kappa)$)	κ_A
@ScopeChange(A)	@ScopeChange(A)	κ_A

Figure A.2: Statement renaming in SES-light (Part 2)

Proof Sketch: By induction on reduction rules. The transition axioms form the base cases which are proven by case analysis. The contextual rules form the inductive case. In order to prove those we prove from the definition of the map k (Figures A.1 and A.2) that for all reduction contexts C, statements s_1, s_2, and scope-binding maps κ_1, κ_2 if $k(s_1, \kappa_1) = k(s_2, \kappa_2)$ then $k(C[s_1], \kappa_1]) = k(C[s_2], \kappa_2)$. The case for contextual rules follows immediately from this result. □

Proposition 8 *For all variable-renaming maps α, statement reduction contexts C, statements s_1, s_2, and scope-binding map κ_1, κ_2, if $k(s_1, \kappa_1) = k(s_2, \kappa_2) = \kappa$ (say) then there exists a context $C_{\alpha,\kappa}$ such that $Rn_s(C[s_1], \alpha, \kappa_1) = C_{\alpha,\kappa}[Rn(s_1, \alpha, \kappa_1)]$ and $Rn_s(C[s_2], \alpha, \kappa_1) = C_{\alpha,\kappa}[Rn(s_2, \alpha, \kappa_2)]$.*

Proof Sketch: We prove this proposition by a case analysis on the set of statement reduction contexts. For each of cases, the proposition follows from the definition of Rn_s from Figures A.1 and A.2. As an example we show the case for the context $C := \_; s$.

By definition of Rn_s, for any two statements s_1, s_2, and scope-binding map κ_1, κ_2 such that $k(s_1, \kappa_1) = k(s_2, \kappa_2) = \kappa$, the following holds:
$Rn_s(C[s_1], \alpha, \kappa_1) = Rn_s(s_1, \alpha, \kappa_1)$; $Rn_s(s, \alpha, k(s_1, \kappa_2)) = Rn_s(s_1, \alpha, \kappa_1)$; $Rn_s(s, \alpha, \kappa)$
$Rn_s(C[s_2], \alpha, \kappa_2) = Rn_s(s_2, \alpha, \kappa_2)$; $Rn_s(s, \alpha, k(s_2, \kappa_2)) = Rn_s(s_1, \alpha, \kappa_1)$; $Rn_s(s, \alpha, \kappa)$
If we let $C_{\alpha,\kappa}$ be the context $\_; Rn_s(s, \alpha, \kappa)$ then we have that
$Rn_s(C[s_1], \alpha, \kappa_1) = C_{\alpha,\kappa}[Rn(s_1, \alpha, \kappa_1)]$ and $Rn_s(C[s_2], \alpha, \kappa_1) = C_{\alpha,\kappa}[Rn(s_2, \alpha, \kappa_2)]$.
Thus the proposition holds for this case. □

Lemma 10 *For all variable-renaming maps α, for all well-formed states S_1, S_2, $S_1 \rightarrow S_2$ iff $Rn(S_1, \alpha) \rightarrow Rn(S_2, \alpha)$.*

Proof Sketch: The proof is carried out by an induction on the set of reduction rules. The base cases are the transition axioms. We prove the base cases by a manual case analysis. As an example we sketch out the proof for the assignment axiom

$$\frac{CanPutVar(A, y) \qquad K, B = Update(H, A, y, v, \{\mathsf{writable}\})}{H, A, y{=}v \rightarrow K, B, \mathsf{N}}$$

Let $H_r, A_r, y_r{=}v$ be the state $Rn(H, A, y{=}v, \alpha, \kappa)$. From the definition of state renaming and *CanPutVar* it follows that $CanPutVar(A, y)$ holds iff $CanPutVar(A_r, y_r)$

holds, since the renaming does not alter the attributes of a variable in an activation record. Furthermore, it can be shown that $K, B = \mathit{Update}(H, A, y, v, \{\mathsf{writable}\})$ iff then $\mathit{Rn}(K, \alpha), \mathit{Rn}(B, \alpha) = \mathit{Update}(H_r, A_r, y_r, v, \{\mathsf{writable}\})$. As a result it follows that $H, A, y{=}v \rightarrow K, B, \mathsf{N}$ iff $H_r, A_r, y_r{=}v \rightarrow K_r, B_r, \mathsf{N}$. This proves the lemma for the assignment rule.

The contextual rules form the inductive case. Given any reduction context C and state $S_1 = H_1, A_1, s_1$, let $C[S_1]$ denote the state $H_1, A_1, C[s_1]$. To prove the inductive case we must show that for all contexts C and states $C[S_1]$, $C[S_2]$, $C[S_1] \rightarrow C[S_2]$ iff $\mathit{Rn}(C[S_1], \alpha) \rightarrow \mathit{Rn}(C[S_2], \alpha)$. From the contextual rules we have that

$$C[S_1] \rightarrow C[S_2] \quad \text{iff} \quad S_1 \rightarrow S_2 \tag{A.14}$$

From the induction hypothesis we have,

$$S_1 \rightarrow S_2 \quad \text{iff} \quad \mathit{Rn}(S_1, \alpha) \rightarrow \mathit{Rn}(S_2, \alpha) \tag{A.15}$$

From Proposition 7, we have $k(\mathit{term}(S_1, \kappa_{\mathit{stack}(S_1)})) = k(\mathit{term}(S_2, \kappa_{\mathit{stack}(S_2)})) = \kappa$. Combining this with Proposition 8 we have that there exists a context $C_{\alpha,\kappa}$ such that

$$\mathit{Rn}_s(C[s_1], \alpha, \kappa_1) = C_{\alpha,\kappa}[\mathit{Rn}(s_1, \alpha, \kappa_1)] \tag{A.16}$$

$$\mathit{Rn}_s(C[s_2], \alpha, \kappa_1) = C_{\alpha,\kappa}[\mathit{Rn}(s_2, \alpha, \kappa_2)] \tag{A.17}$$

From the contextual rules we have that

$$\mathit{Rn}(S_1, \alpha) \rightarrow \mathit{Rn}(S_2, \alpha) \quad \text{iff} \quad C_{\alpha,\kappa}[\mathit{Rn}(s_1, \alpha, \kappa_1)] \rightarrow C_{\alpha,\kappa}[\mathit{Rn}(s_2, \alpha, \kappa_2)] \tag{A.18}$$

From Conditions A.14, A.15, A.16, A.17 and A.18, it follows that $C[S_1] \rightarrow C[S_2]$ holds iff $\mathit{Rn}(C[S_1], \alpha) \rightarrow \mathit{Rn}(C[S_2], \alpha)$, thus proving the inductive case. □

Restatement of Theorem 9. *For all well-formed states S, $\mathit{Rn}(\tau(S)) = \tau(\mathit{Rn}(S))$*

Proof Sketch: Using Lemma 10, the theorem can be proven by a straightforward induction on the length of reduction traces. □

A.5 Proofs from Chapter 8

In this section we prove Theorem from Chapter 8 using the SES-light analysis framework developed in Section 7.3.

A.5.1 Preliminaries

In order to prove Theorem , we define a map $Enc : 2^{\Sigma} \rightarrow 2^{Facts}$ (abstraction map) for encoding a set of program states as a set of Datalog facts. We then show the following: (1) Given an API implementation t, the encoding of the initial set of program states ($\mathcal{S}_0(t)$) is over-approximated by the initial set of facts ($\mathcal{F}_0(t)$) provided to the Datalog solver, (2) For any set of program states $\mathcal{S}$, encoding of the set of all states reachable from states in $\mathcal{S}$) is over-approximated by the set of all consequence facts derived from the encoding of $\mathcal{S}$, and (3) the points-to map $PtsTo$ is over-approximated by the abstract points-to map $PtsTo_{\mathcal{D}}$, under the encoding. Property (1) is shown by Lemma 11, property (2) by Lemma 12, and property (3) by Lemma 13.

Encoding of States. We define Enc by individually encoding the heap, stack and term part of each state in the set. Encoding of terms is carried out using the map Enc_t defined in Figures 8.2, 8.3, 8.4 and 8.5. In order to encode the heap and stack, we define two auxiliary maps Enc_H and Enc_A respectively.

Given a stack A, we define $Enc_A(A)$ as the union of the following sets of facts: (1) set of facts $Stack(x, \hat{l})$ for which there exists a record $R \in A$, such that $R \neq ERG$, R is not allocated by an $\widehat{\mathsf{eval}}_{\mathsf{nf}}$ statement, and that satisfies $x \in dom(R)$ and $Lab(R(x)) = \hat{l}$, and (2) set of facts $Stack(\alpha(\mathsf{x}_{\mathsf{eval}}, Lab(R)), \hat{l})$ for each variable record $R \in A$, allocated by an $\widehat{\mathsf{eval}}_{\mathsf{nf}}$ call, that satisfies $Lab(R(x)) = \hat{l}$ for some variable x.

Given a heap H, $Enc_H(H)$ is defined as the union of the following sets of facts

(1) Set of facts $Heap(Lab(l_1), x, Lab(l_2))$ for all locations l_1, l_2 and property name x for which $H(l_1).x = l_2$ holds and $l_1 \notin dom(H_0)$ (in other words l_1 is not a *built-in* location).

(2) Set of facts $Enc_t(fv, Lab(B)) \cup Enc_A(B)$ and $\{FuncType(Lab(l))\}$, for all closures fv, B for which there exists a locations $l \notin dom(H_0)$ and $H(l).\mathsf{@code} =$

fv, B.

(3) Set of facts $Prototype(Lab(l_1), Lab(l_2))$ for all locations l_1, l_2 such that $H(l_1).\texttt{@proto} = l_2$ and $l_1 \notin dom(H_0)$.

(4) Set of facts $Global(p)$ and $Stack(p, Lab(l))$ for all user-properties p and locations l such that $H(\#global).p = l$.

(5) Set of facts $NotBuiltin(Lab(l))$ for all locations $l \in dom(H) \setminus dom(H_0)$.

(6) The initial encoding $\mathcal{I}_0$ of the built-in objects.

We now formally define the map Enc as follows:

Definition 29 (Enc) *Given a set of states $\mathcal{S} \in 2^{\Sigma}$, $Enc(\mathcal{S})$ is defined as $\bigcup_{(H,A,t)\in\mathcal{S}} Enc_H(H) \cup Enc_A(A) \cup Enc_t(t, \hat{l}_g)$.*

A.5.2 Main Results

We first prove that the encoding of the initial set of program states $\mathcal{S}_0(t)$ is over-approximated by the initial set of facts $\mathcal{F}_0(t)$. The definitions of $\mathcal{S}_0(t)$ and $\mathcal{F}_0(t)$ are provided in Section 8.2.1 and Figure 8.7 respectively.

Proposition 9 $Enc_H(H_0) \cup Enc_A(A_0) \subseteq \mathcal{I}_0$.

Proof Sketch: The initial stack A_0 only contains the global record ERG. Therefore by definition $Enc_A(A_0) = \emptyset$. From the definition of Enc_H, we have that $Enc_H(H_0)$ is the union of (1) set of facts $Global(p)$ for all properties $p \in dom(H(\#global))$, and (2) $\mathcal{I}_0$. By definition of $\mathcal{I}_0$ (see Section 8.2.1) set (1) is a subset of $\mathcal{I}_0$. Therefore $Enc_H(H_0) \subseteq \mathcal{I}_0$ and the proposition holds. □

Lemma 11 *For all statements $t \in$ SES-light, $Enc(\mathcal{S}_0(t)) \subseteq \mathcal{F}_0(t)$.*

Proof Sketch: By definition, $\mathcal{S}_0(t) := \{H_0, A_0, \mathsf{SYS}(t, s, \mathsf{api}, \mathsf{un}) \mid s \in Stmts^{\mathrm{SESl}}_{user}\}$. Hence, $Enc(\mathcal{S}_0(t)) = Enc_H(H_0(t)) \cup Enc_A(\mathcal{S}_0(t)) \cup \{Enc_t(\mathsf{SYS}(t, s, \mathsf{api}, \mathsf{un}), \hat{l}_g) \mid s \in Stmts^{\mathrm{SESl}}_{user}\}$.

By definition, $\mathcal{F}_0(t) = Enc_t(\mathsf{SYS}(t, s, \mathsf{api}, \mathsf{un}), \hat{l}_g) \cup \mathcal{I}_0$ for an arbitrary $s \in Stmts_{user}^{\mathrm{SESl}}$. Since from Proposition 9, $Enc_H(H_0) \cup Enc_A(A_0) \subseteq \mathcal{I}_0$, all we need to show is that $Enc_t(\mathsf{SYS}(t, s, \mathsf{api}, \mathsf{un}), \hat{l}_g)$ is the same for all $s \in Stmts_{user}^{\mathrm{SESl}}$. This follows from the definitions of Enc_t and $\mathsf{SYS}(t, s, \mathsf{api}, \mathsf{un})$ (owing to the fact that the encoding of $\mathsf{eval}_{\mathsf{nf}}$ statements is independent of the term being eval-ed). Thus $Enc(\mathcal{S}_0(t)) \subseteq \mathcal{F}_0(t)$. □

Let $\mathcal{R}$ be the inference rules defined in Figure 8.6. We prove a lemma that says that the encoding of the set of all states reachable from $\mathcal{S}$, is over-approximated by the set of all consequence facts derivable from the encoding of $\mathcal{S}$. In order to prove this lemma, we first extend the term encoding map Enc_t to internal terms.

Recall that in the labeled semantic of SES-light, all statements appearing in a term carry labels. Moreover the label associated with the internal statements are the labels of the user-statement that created them. Therefore augment the semantics so that all internal statement appearing in a state reduction are always marked with the internal statement that created them. For example in the reduction,
$H, A, y{=}\ v_1[v_2,an] \rightarrow H, A, \mathtt{@TO}(@_1,v_1);\mathtt{@TS}(@_2,v_2);\ \mathsf{y} = @_1[@_2,\mathsf{an}]$, the internal statements $\mathtt{@TO}(@_1,v_1)$, $\mathtt{@TS}(@_2,v_2)$ and $\mathsf{y} = @_1[@_2,\mathsf{an}]$ are labeled with the statement $y{=}\ v_1[v_2,an]$. We use $SLab(s)$ as the user-statement associated with an internal statement s.

The encoding of an internal statement s under a enclosing-scope label $\hat{l}$, is defined as the union of the following: (1) $Enc_t(s, \hat{l})$, and (2) $\{Enc_t(s_1, \hat{l}) \mid s_1$ is nested in $s\}$. For example, the encoding of the internal statement $s := \mathtt{@Fun3}(\mathsf{tag},\ A,\ x,\ v,\ s_1)$ under a enclosing-scope label $\hat{l}$, is $Enc_t(SLab(s), \hat{l}) \cup Enc_t(s_1, \hat{l})$. In the rest of the results, we assume that the statement encoding map Enc_t (and transitively the state encoding map Enc) applies to internal statement as well.

Proposition 10 *Given a statement reduction context C and a label $\hat{l}$, there exists a set of facts $\mathcal{F}_C$ such that for all statement s, $Enc_t(C[s], \hat{l}) := Enc_t(s, \hat{l}) \cup \mathcal{F}_C$.*

Proof Sketch: By straightforward case analysis on the set of statement reduction contexts. □

In order to state the next proposition, we introduce the following notation: for any monotone self map F on a complete lattice, and an element x on the lattice, we use

$lfp_x F$ to denote the least fixed point of F greater than x. The existence of such a fixed point is guaranteed by Tarski's fixedpoint theorem. ([16]). Observe that the powersets 2^{Σ} and 2^{Facts} form complete lattices under the natural subset order.

Proposition 11 *Consider a power set lattice $2^{\mathcal{S}}$ and a monotone self-map f over the lattice. The following holds:*

(1) *If f is continuous then $lfp_x f := \bigcup\{f^n(x) \mid n \geq 1\}$.*

(2) *If $\mathcal{S}$ is finite then $lfp_x f \bigcup\{f^n(x) \mid n \geq 1\}$.*

Proof Sketch: The proof of the first-part is by Kleene's fixed point theorem(see [14]). The proof of second-part is as follows: Since $\mathcal{S}$ is finite, the set $\{f^n(x) \mid n \geq 1\}$ is finite. Therefore there exists an $n = a$ such that $f^{a+1}(x) = f^a(x)$ and therefore $f^(a)(x)$ is a fixed point of f. By monotonicity of f, it follows that $\bigcup\{f^n(x) \mid n \geq 1\} = f^a(x)$, and thus $\bigcup\{f^n(x) \mid n \geq 1\}$ is a fixed-point of f. Now all that remains to show is that this is the least fixed-point above x. This property can be proven by contradiction. Suppose there exists a fixed-point y such that $x \subseteq y \subseteq f^a(x)$. By monotonicity, we have $f^a \subseteq f^a(y) = y$, which is a contradiction. □

Lemma 12 *For all set of states $\mathcal{S} \in 2^{States}$, $Enc(Reach(\mathcal{S})) \subseteq Cons(Enc(\mathcal{S}), \mathcal{R})$.*

Proof Sketch: Given an element $\mathcal{S} \in 2^{\Sigma}$, we define the concrete single-step evaluation map $N_{\rightarrow}(\mathcal{S})$ as $\mathcal{S} \cup \{S' \mid \exists S \in \mathcal{S} : S \rightarrow S'\}$. It is easy to see that $Reach(\mathcal{S}) = lfp_{\mathcal{S}} N_{\rightarrow}$. Given an element $\mathcal{F} \in 2^{Facts}$, we define the abstract single-step evaluation map $N_{\mathcal{D}}(\mathcal{F})$ as $\mathcal{F} \cup Infer_1(\mathcal{F}, \mathcal{R})$ where $Infer_1(\mathcal{F}, \mathcal{R})$ is the set of facts obtained by applying the rules $\mathcal{R}$ exactly once[1]. Under the Herbrand semantics of Datalog, $Cons(Enc(\mathcal{S}), \mathcal{R}) = lfp_{Enc(\mathcal{S})} N_{\mathcal{D}}$. Consider the following property:

> *Property A.* For all states S, there exists $n \geq 1$ such that:
> $Enc(N_{\rightarrow}(\{S\})) \subseteq N^n_{\mathcal{D}}(Enc(\{S\}))$.

[1]This is also known as the *elementary production principle.* (see [12])

Observe that the set *Facts* is finite, the map $N_{\rightarrow}$ is continuous and the map $N_{\mathcal{D}}$ is monotonic. Using the fact that the map *Enc* is defined point-wise on a set of states, one can show that Property A implies that $Enc(lfp_{\mathcal{S}} N_{\rightarrow}) \subseteq lfp_{Enc(\mathcal{S})} N_{\mathcal{D}}$. Thus to prove that lemma, all that remains to show is Property A.

We prove Property A by an induction on the set of reduction rules. The transition axioms form the base cases which are proven by a straightforward case analysis. The contextual rules form the inductive case. For a reduction context C and any state $S := H, A, s$, we define $C[S] = H, A, C[s]$. It is suffices to show that $Enc(\{N_{\rightarrow}(C[S])\}) \subseteq N^n_{\mathcal{D}}(Enc(\{C[S]\}))$ for some n. By contextual rules, $N_{\rightarrow}(C[S]) = C(N_{\rightarrow}[S])$. Using the definition of *Enc* and Proposition 10, one can show that there exists a set of facts F_C such that $Enc(C[N_{\rightarrow}(S)]) = F_C \cup Enc(\{N_{\rightarrow}(S)\})$ and $Enc(C(N_{\rightarrow}(S)]) = F_C \cup Enc(\{N_{\rightarrow}(S)\})$ By induction hypothesis, $Enc(\{N_{\rightarrow}(S)\}) \subseteq N^n_{\mathcal{D}}(Enc(\{S\}))$ for some n. By definition of $N_{\mathcal{D}}$, it follows that $F_C \cup N^n_{\mathcal{D}}(Enc(\{S\})) \subseteq N^n_{\mathcal{D}}(F_C \cup Enc(\{S\})) = N^n_{\mathcal{D}}(Enc(\{C[S]\}))$. The inductive case follows immediately from this. □

The final lemma for proving soundness is that the abstract points-to map $PtsTo_{\mathcal{D}}$ safely over-approximates the concrete points-to map *PtsTo*, under the encoding *Enc*.

Lemma 13 *For all $v \in Vars$ and set of states $\mathcal{S} \in 2^{\Sigma}$,*
$PtsTo(v, \mathcal{S}) \subseteq PtsTo_{\mathcal{D}}(v, Enc(\mathcal{S}))$.
Proof Sketch: By definition of *Enc*, for all states $H, A, t \in \mathcal{S}$, for all user-variables v and for all locations l, if $H(\#global).\text{“v”} = l$ then $Stack(v, Lab(l)) \in Enc(\mathcal{S})$. The lemma follows immediately from this property. □

Restatement of Theorem A.5.1. *For all statements t and security-critical object labels L_{sec}, $\mathcal{D}(t, L_{sec}) \implies Confine(t, L_{sec})$.*
Proof Sketch: From Figure 8.7,
$\mathcal{D}(t, L_{sec}) \Leftrightarrow PtsTo_{\mathcal{D}}(\text{“un”}, Cons(\mathcal{F}_0(t), \mathcal{R})) \cap L_{sec} = \emptyset$.
From monotonicity of *Cons* and $PtsTo_{\mathcal{D}}$ and Lemmas 12, 11, 13, it follows that the set $PtsTo(\text{“un”}, Reach(\mathcal{S}_0(t)))$ is a subset of $PtsTo_{\mathcal{D}}(\text{“un”}, Cons(\mathcal{F}_0(t), \mathcal{R}))$. The theorem follows immediately from this result. □

Bibliography

[1] Ecma international. `http://www.ecma-international.org/`.

[2] Yelp. `http://www.yelp.com/`.

[3] B. Adida. BeamAuth: Two-factor Web authentication with a Bookmark. In *ACM Conference on Computer and Communications Security*, 2007.

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[5] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *European Conference on Object-Oriented Programming*, 2005.

[6] I. Atsushi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Transactions on Programming Languages and Systems*, 1999.

[7] A. Barth, C. Jackson, and W. Li. Attacks on JavaScript mashup communication. In *Web 2.0 Security & Privacy (W2SP)*, 2009.

[8] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[9] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *ACM Conference on Computer and Communications Security*, 2010.

[10] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8, 1990.

[11] Google Caja Team. Google-Caja: A source-to-source translator for securing JavaScript-based Web content. `http://code.google.com/p/google-caja/`.

[12] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1, 1989.

[13] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN Conference on Principles of Programming Languages*, 1977.

[15] D. Crockford. ADsafe: Making JavaScript safe for advertising. `http://www.adsafe.org/`, 2008.

[16] F. Echenique. A short and constructive proof of Tarski's fixed-point theorem. *International Journal of Game Theory*, 33, 2005.

[17] U. Erlingsson. *A New Approach to Mobile Code Security*. PhD thesis, Princeton University, 1999.

[18] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2003.

[19] U. Erlingsson and F. B. Schneider. Irm enforcement of Java stack inspection. In *IEEE Symposium on Security & Privacy*, 2000.

[20] D. Evans and A. Tmyman. Flexible policy-directed code safety. In *IEEE Symposium on Security & Privacy*, 1999.

[21] OpenSocial Foundation. OpenSocial. `http://www.opensocial.org/`.

[22] P. Gardner, G. Smith, M. Wheelhouse, and U. Zarfaty. Local hoare reasoning about DOM. In *ACM Conference on Principles of Distributed Systems*, 2008.

[23] L. Gong, M. Mueller, and H. Prafullch. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[24] Google. iGoogle. `http://www.google.com/ig`.

[25] S. Guarnieri and B. V. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, 2009.

[26] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE Symposium on Security & Privacy*, 2011.

[27] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming*, 2010.

[28] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming Languages*, 2011.

[29] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *IEEE Symposium on Computer Security Foundations*, 2012.

[30] D. Herman. Classic JavaScript. `http://www.ccs.neu.edu/home/dherman/javascript/`.

[31] D. Van Horn and H. G. Mairson. Deciding kCFA is complete for EXPTIME. In *ACM SIGPLAN International Conference on Functional Programming*, 2008.

[32] ECMA International. ECMAScript language specification. stardard ECMA-262, 3rd Edition, Dec 1999.

[33] ECMA International. ECMAScript language specification. stardard ECMA-262, 5th Edition, Dec 1999.

[34] C. Jackson and H. Wang. Subspace: Secure cross-domain communication for Web mashups. In *International World Wide Web Conference*, 2007.

[35] D. Jang and K. Choe. Points-to analysis for JavaScript. In *Annual Computer Security Applications Conference*, 2009.

[36] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *International World Wide Web Conference*, 2007.

[37] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting Web application vulnerabilities (short paper). In *IEEE Symposium on Security & Privacy*, 2006.

[38] J. H. Morris Jr. Protection in programming languages. *Communications of the ACM*, 16, 1973.

[39] F. De Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. Smash: Secure component model for cross-domain mashups on unmodified browsers. In *International World Wide Web Conference*, 2008.

[40] S. Krishnamurthi. Confining the ghost in the machine: Using types to secure JavaScript sandboxing. In *Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications*, 2010.

[41] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 1973.

[42] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[43] B. V. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, 2005.

[44] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on Web advertisements. In *USENIX Security Symposium*, 2010.

[45] S. Maffeis, J. C. Mitchell, and A. Taly. Complete ECMA 262, 3^{rd} edition operational semantics. `http://theory.stanford.edu/~ataly/Semantics/es3Semantics.txt`. See also: `http://jssec.net/semantics/`.

[46] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems*, 2008.

[47] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *European Symposium on Research in Computer Security*, 2009.

[48] S. Maffeis, J. C. Mitchell, and A. Taly. Run-time enforcement of untrusted JavaScript subsets. In *Web 2.0 Security & Privacy (W2SP)*, 2009.

[49] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted Web applications. In *IEEE Symposium on Security & Privacy*, 2010.

[50] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *IEEE Symposium on Computer Security Foundations*, 2009.

[51] J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *ACM Symposium on Information, Computer and Communications Security*, 2010.

[52] J. Magazinius, P. H. Phung, and D. Sands. Safe wrappers and sane policies for self protecting JavaScript. In *Nordic Conference on Secure IT Systems*, 2010.

[53] A. Mettler and D. Wagner. The Joe-E language specification (draft). Technical Report UCB/EECS-2006-26, U.C. Berkeley, May 2006.

[54] L. A. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symposium on Security and Privacy*, 2010.

[55] J. Midtgaard and T. Jensen. A calculational approach to control-flow analysis by abstract interpretation. In *International Static Analysis Symposium*, 2008.

[56] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[57] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transaction on Software Engineering Methodologies*, 14(1), 2005.

[58] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[59] M. S. Miller, C. Morningstar, and B. Frantz. Capability-based financial instruments. In *International Conference on Financial Cryptography and Data Security*, 2001.

[60] M. S. Miller, K.-P. Yee, and J.S. Shapiro. Capability myths demolished. Technical report, Johns Hopkins University, 2003.

[61] Mark S. Miller. initSES.js. `http://code.google.com/p/es-lab/source/browse/trunk/src/ses/`, 2010.

[62] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *ACM SIGPLAN Conference on Principles of Programming Languages*, 1998.

[63] Mozilla. Content security policy. `http://http://people.mozilla.org/~bsterne/content-security-policy/`, 2011.

[64] T. Murray and G. Lowe. Analysing the information flow properties of object-capability patterns. In *USENIX Conference on File and Storage Technologies*, 2009.

[65] G. C. Necula. Proof-carrying code. In *ACM SIGPLAN Conference on Principles of Programming Languages*, 1997.

[66] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self protecting JavaScript. In *ACM Symposium on Information, Computer and Communications Security*, 2009.

[67] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. In *ACM Symposium on Information, Computer and Communications Security*, 2009.

[68] G. J. Politz, S. A. Aliopoulos, A. Guha, and S. Krishnamurthi. Adsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security Symposium*, 2011.

[69] J. A. Rees. A security kernel based on the lambda-calculus. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1996.

[70] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. In *USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[71] E Rights. The E language. `http://erights.org/elang/index.html`.

[72] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21, 2003.

[73] M. I. Seltzer, E. Yasuhiro, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *USENIX Symposium on Operating Systems Design and Implementation*, 1996.

[74] J. S. Shapiro and S. Weber. Verifying the eros confinement mechanism. In *IEEE Symposium on Security & Privacy*, Washington, DC, USA, 2000. IEEE Computer Society.

[75] J. Siek and W. Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, 2007.

[76] F. Spiessens. *Patterns of Safe Collaboration*. PhD thesis, Université Catholique de Louvain, 2007.

[77] F. Spiessens and P. Van Roy. The Oz-E Project: Design guidelines for a secure multiparadigm programming language. In *Multiparadigm Programming in Mozart/OZ*, volume 3389. Springer Verlag, 2005.

[78] M. Stiegler. Emily: A high performance language for enabling secure cooperation. In *International Conference on Creating, Connecting and Collaborating through Computing*, 2007.

[79] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Complete operational semantics for SES-light. `http://theory.stanford.edu/~ataly/Semantics/seslSemantics.txt`.

[80] A. S. Tanenbaum, R. Van Renesse, H. Van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. Van Rossum. Experiences with the amoeba distributed operating system. *Communications of the ACM*, 33, 1990.

[81] The Facebook Team. Facebook. `http://www.facebook.com/`.

[82] The Facebook Team. FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`.

[83] P. Thiemann. Towards a type system for analyzing JavaScript programs. In *European Symposium on Programming Languages*, 2005.

[84] P. Thiemann. A type safe DOM API. In *International Symposium on Database Programming Languages*, 2005.

[85] D. Vardoulakis and O. Shivers. CFA2: A context-free approach to control-flow analysis. In *European Symposium on Programming Languages*, 2010.

[86] W3C. Document object model (dom). `http://www.w3.org/DOM/DOMTR`, 2004.

[87] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles*, 1994.

[88] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architecture for Java. In *ACM Symposium on Operating Systems Principles*, 1997.

[89] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *IEEE Symposium on Security & Privacy*, 1998.

[90] J. Whaley. BDDBDDB: BDD based deductive database. `http://bddbddb.sourceforge.net/`, 2004.

[91] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004.

[92] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, 2006.

[93] Yahoo! Inc. Yahoo! Application Platform. `http://developer.yahoo.com/yap/`.

[94] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *ACM SIGPLAN Conference on Principles of Programming Languages*, 2007.