Building Secure Web Applications: From Theory to Practice

Abstract

Web pages are the front door to almost any online service. Despite their success, we constantly see vulnerabilities being exposed in web sites. The reason for that are commonly programming errors leading to serious security breaches—this is not surprising given the complexity of web applications (web apps). The status quo security practices consists on mainly add-hoc solutions. In this course, we present a disciplined manner to avoid such programming errors.

Information-Flow Control (IFC) and Mandatory Access Control (MAC) emerge as promising technologies to harden web apps. To avoid information leaks (data corruption), IFC and MAC systems restrict programmers from building web sites which irresponsibly distribute (modifies) sensitive (trustworthy) data . The course introduces security problems behind web apps, the foundations for IFC and MAC as well as their applicability to online systems. The material presented is based on recent research results.

Course resources

Web page: `http://www.cse.chalmers.se/~russo/eci2015/`

Twitter: `@russoECI2015`

Lecturer

Alejandro Russo is an associate professor at the *Chalmers University of Technology* in Göteborg, Sweden (where he did his PhD). Previously, he has been a research assistant at the *Stevens Institute of Technology* (2015, USA), visiting assistant professor at *University of Buenos Aires* (2011, Argentina), and twice visiting associate professor at *Stanford University* (2013, 2014-2015, USA). He is an expert in programming languages analysis for security. He specializes in information-flow control (IFC), where the protection of confidentiality and integrity of data is of ultimate importance. His work ranges from theoretical foundations to the concrete implementation of systems.

Introduction to Web Security

Building Secure Web Applications

Alejandro Russo
russo@chalmers.se
ECI 2015, UBA, Buenos Aires, Argentina

Hacker finds vulnerability in Facebook, can delete your photo albums

Comments considered harmful: WordPress web hijack bug revealed

Patch NOW after researcher drops zero-day on popular blog software

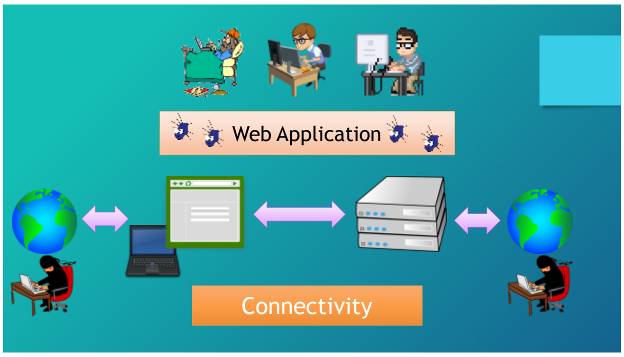
Google fixed a flaw in YouTube that let people delete anybody's videos

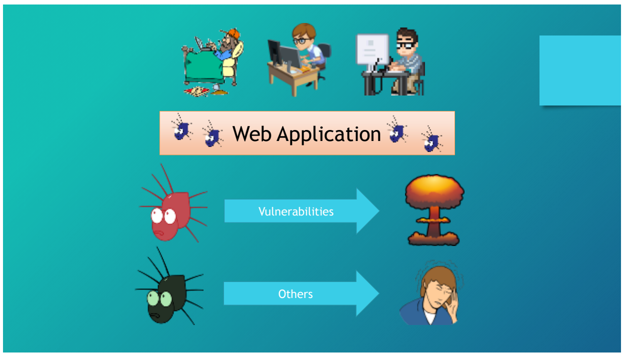
FORBES 5/06/2015 @ 6:01AM | 15,359 views

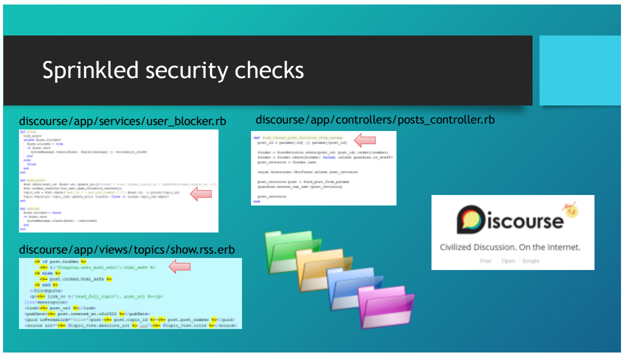
Meet The Darpa-Backed Hackers Building A Google For Every Web Weakness

AirDroid fixes flaw that could grant hackers full control of your phone

2015 News!







Sprinkled security checks

discourse/app/services/user_blocker.rb

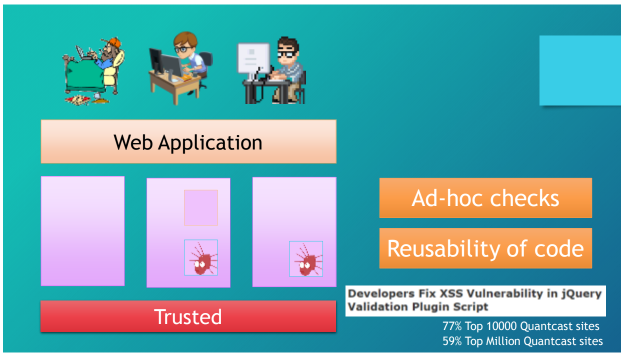
```
def block
  hide_posts
  unless @user.blocked?
    @user.blocked = true
    if @user.save
      SystemMessage.create(@user, @opts[:message] || :blocked_by_staff)
    end
  else
    false
  end
end

def hide_posts
  Post.where(user_id: @user.id).update_all(["hidden = true, hidden_reason_id = [illegible]
  Post.hidden_reasons[:new_user_spam_threshold_reached]
  topic_ids = Post.where('user_id = ? and post_number = ?', @user.id, 1).pluck(:topic_id)
  Topic.where(id: topic_ids).update_all({ visible: false }) unless topic_ids.empty?
end

def unblock
  @user.blocked = false
  if @user.save
    SystemMessage.create(@user, :unblocked)
  end
end
```

discourse/app/controllers/posts_controller.rb

```
def find_latest_post_revision_from_params
  post_id = params[:id] || params[:post_id]

  finder = PostRevision.where(post_id: post_id).order(:number)
  finder = finder.where(hidden: false) unless guardian.is_staff?
  post_revision = finder.last

  raise Discourse::NotFound unless post_revision

  post_revision.post = find_post_from_params
  guardian.ensure_can_see!(post_revision)

  post_revision
end
```

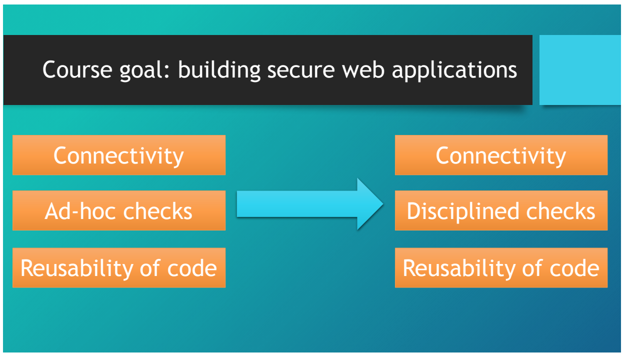
discourse/app/views/topics/show.rss.erb

```
      <% if post.hidden %>
        <%= t('flagging.user_must_edit').html_safe %>
      <% else %>
        <%= post.cooked.html_safe %>
      <% end %>
    </blockquote>
    <p><%= link_to t('read_full_topic'), post_url %></p>
  ]]></description>
  <link><%= post_url %></link>
  <pubDate><%= post.created_at.rfc2822 %></pubDate>
  <guid isPermaLink="false">post-<%= post.topic_id %>-<%= post.post_number %></guid>
  <source url="<%= @topic_view.absolute_url %>.rss"><%= @topic_view.title %></source>
```



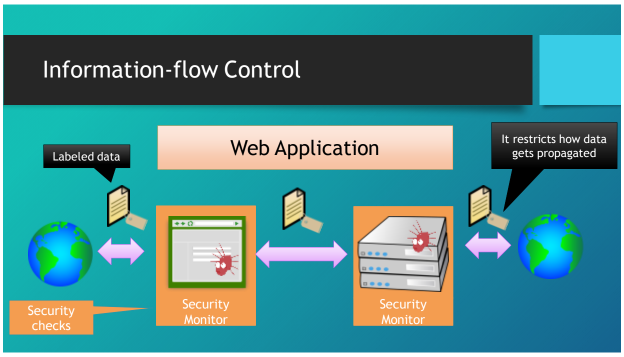
Developers Fix XSS Vulnerability in jQuery Validation Plugin Script

77% Top 10000 Quantcast sites
59% Top Million Quantcast sites

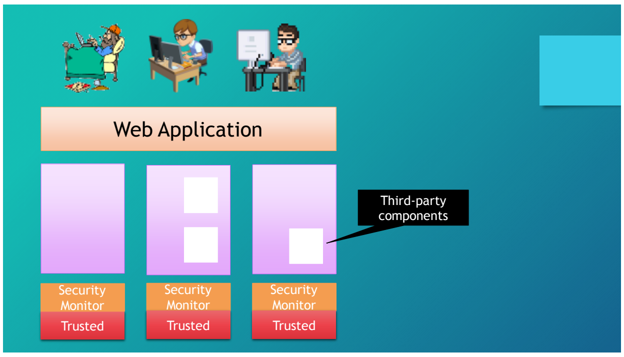


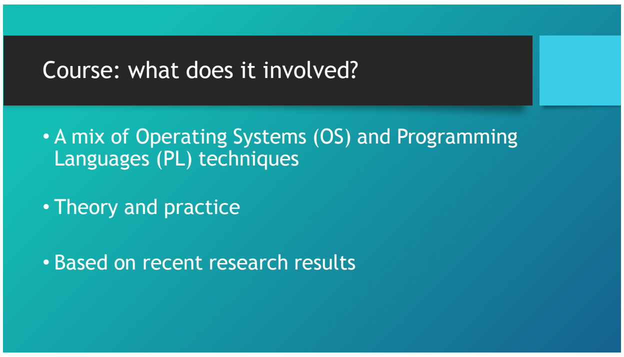
Course goal: building secure web applications

Connectivity
Ad-hoc checks
Reusability of code
Connectivity
Disciplined checks
Reusability of code



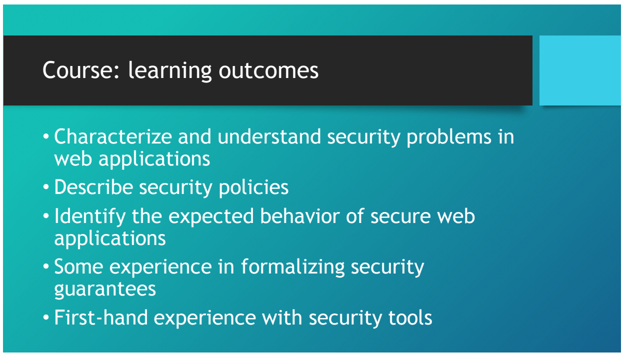
Information-flow Control





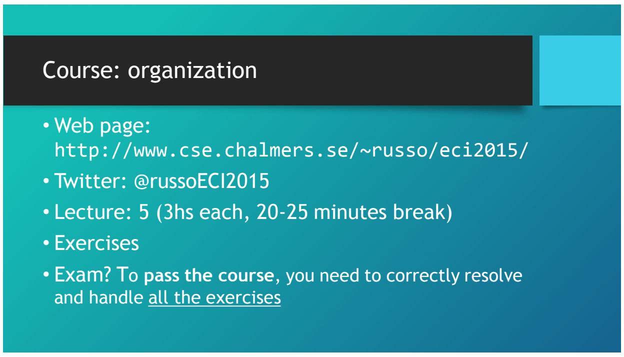
Course: what does it involved?

- A mix of Operating Systems (OS) and Programming Languages (PL) techniques
- Theory and practice
- Based on recent research results



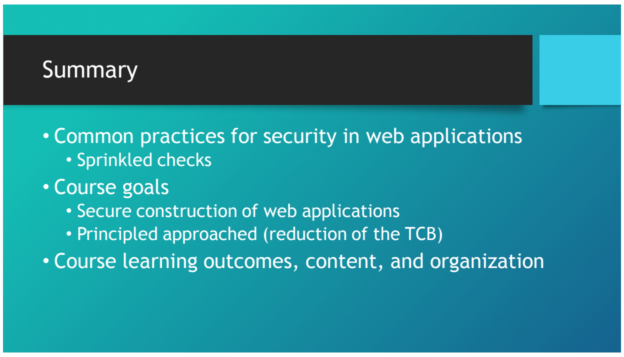
Course: learning outcomes

- Characterize and understand security problems in web applications
- Describe security policies
- Identify the expected behavior of secure web applications
- Some experience in formalizing security guarantees
- First-hand experience with security tools



Course: organization

- Web page: `http://www.cse.chalmers.se/~russo/eci2015/`
- Twitter: @russoECI2015
- Lecture: 5 (3hs each, 20-25 minutes break)
- Exercises
- Exam? To **pass the course**, you need to correctly resolve and handle all the exercises

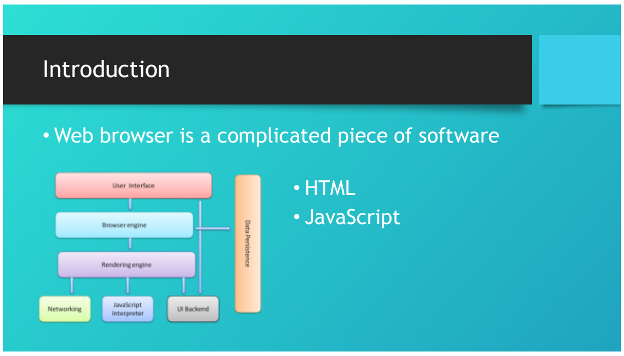


Summary

- Common practices for security in web applications
  - Sprinkled checks
- Course goals
  - Secure construction of web applications
  - Principled approached (reduction of the TCB)
- Course learning outcomes, content, and organization

Security in Web Browsers

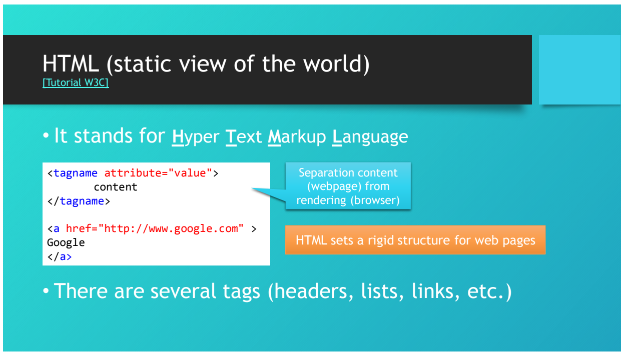
Alejandro Russo
russo@chalmers.se
ECI 2015, UBA, Buenos Aires, Argentina



Introduction

- Web browser is a complicated piece of software

- HTML
- JavaScript



HTML (static view of the world)

[Tutorial W3C]

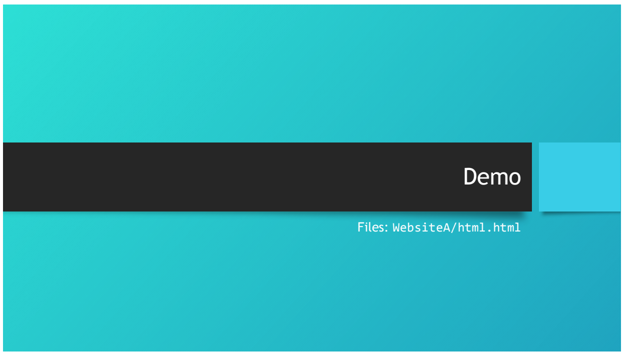
- It stands for Hyper Text Markup Language

```
<tagname attribute="value">
        content
</tagname>

<a href="http://www.google.com" >
Google
</a>
```

Separation content (webpage) from rendering (browser)

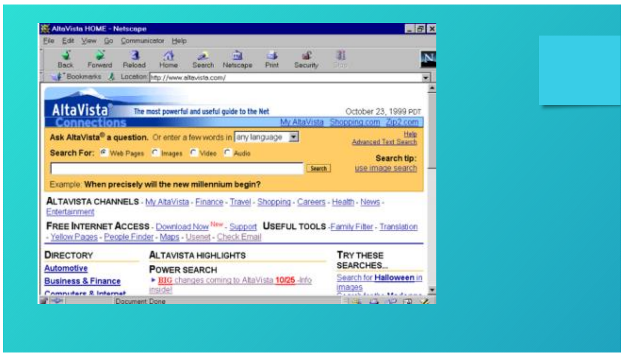
HTML sets a rigid structure for web pages

- There are several tags (headers, lists, links, etc.)



Demo

Files: `WebsiteA/html.html`



AltaVista HOME - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: http://www.altavista.com/
Document: Done

AltaVista® Connections The most powerful and useful guide to the Net October 23, 1999 PDT

My AltaVista Shopping.com Zip2.com

Ask AltaVista® a question. Or enter a few words in any language

Help
Advanced Text Search

Search For: ● Web Pages ○ Images ○ Video ○ Audio

Search

Search tip: use image search

Example: When precisely will the new millennium begin?

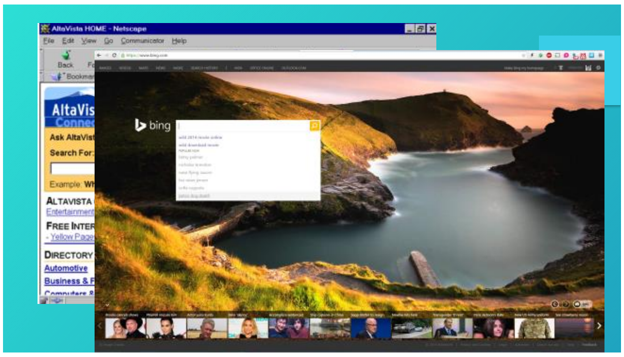
ALTAVISTA CHANNELS - My AltaVista - Finance - Travel - Shopping - Careers - Health - News - Entertainment

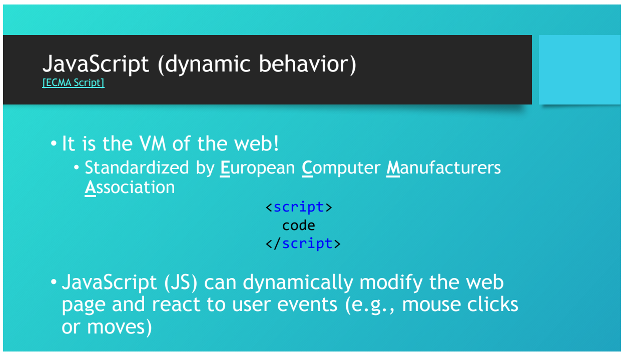
FREE INTERNET ACCESS - Download Now New - Support USEFUL TOOLS - Family Filter - Translation - Yellow Pages - People Finder - Maps - Usenet - Check Email

DIRECTORY
Automotive
Business & Finance
Computers & Internet

ALTAVISTA HIGHLIGHTS
POWER SEARCH
▸ BIG changes coming to AltaVista 10/25 -Info inside!

TRY THESE SEARCHES...
Search for Halloween in images





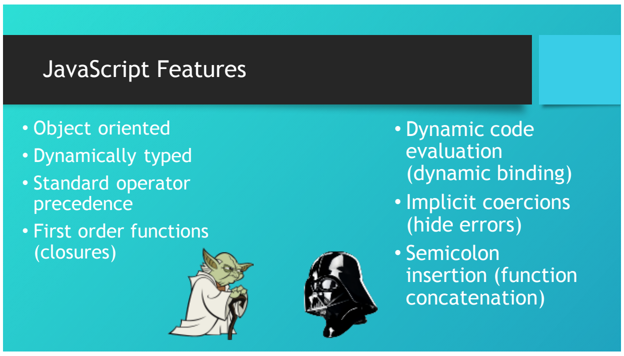
JavaScript (dynamic behavior)

[ECMA Script]

- It is the VM of the web!
  - Standardized by European Computer Manufacturers Association

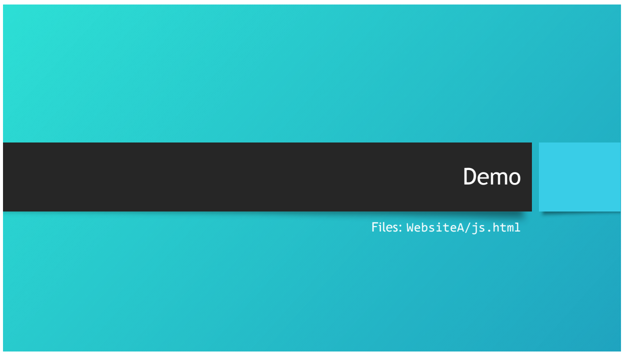
```
<script>
code
</script>
```

- JavaScript (JS) can dynamically modify the web page and react to user events (e.g., mouse clicks or moves)



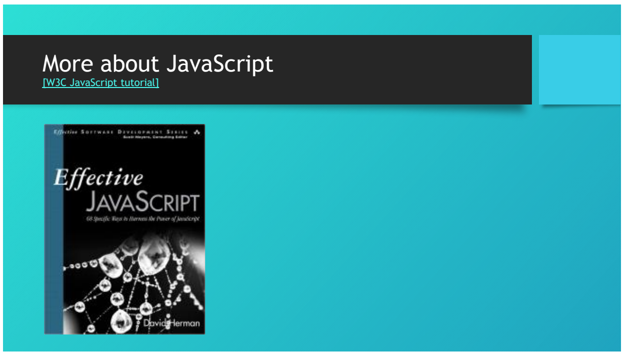
JavaScript Features

- Object oriented
- Dynamically typed
- Standard operator precedence
- First order functions (closures)
- Dynamic code evaluation (dynamic binding)
- Implicit coercions (hide errors)
- Semicolon insertion (function concatenation)



Demo

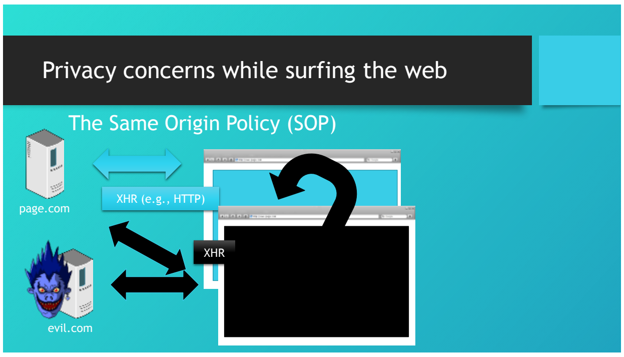
Files: `WebsiteA/js.html`



More about JavaScript

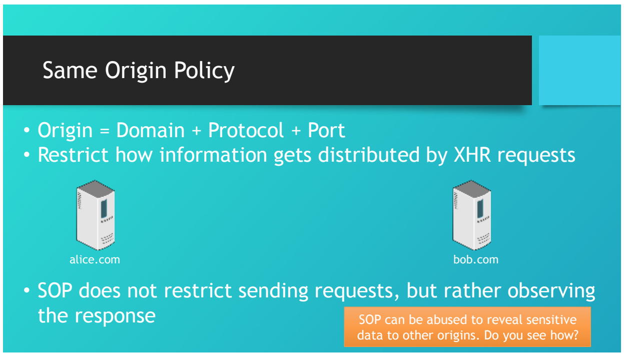
[W3C JavaScript tutorial]

Effective JavaScript
David Herman



Privacy concerns while surfing the web

The Same Origin Policy (SOP)



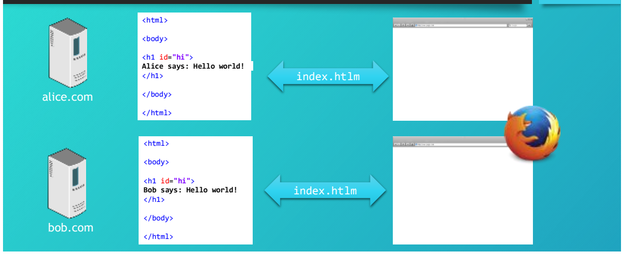
Same Origin Policy

- Origin = Domain + Protocol + Port
- Restrict how information gets distributed by XHR requests

- SOP does not restrict sending requests, but rather observing the response

SOP can be abused to reveal sensitive data to other origins. Do you see how?

Same Origin Policy – Running Example



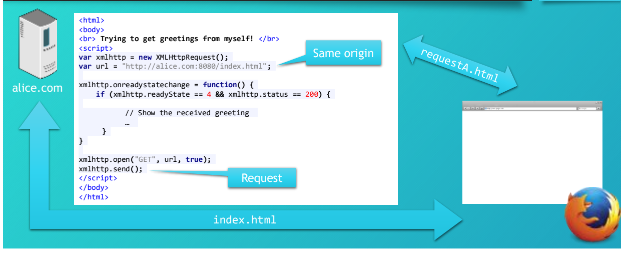
```
<html>

<body>

<h1 id="hi">
Alice says: Hello world!
</h1>

</body>

</html>
```

```
<html>

<body>

<h1 id="hi">
Bob says: Hello world!
</h1>

</body>

</html>
```

Demo

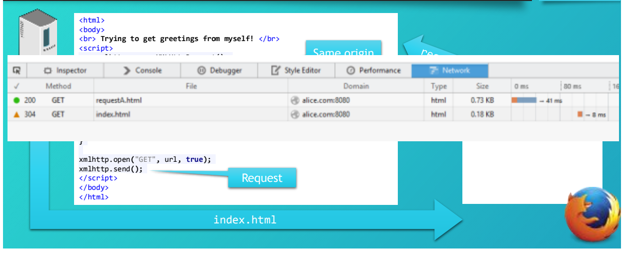
Files: `WebsiteA/index.html` and `WebsiteB/index.html`

Same Origin Policy – Same Origin request



```
<html>
<body>
<br> Trying to get greetings from myself! </br>
<script>
var xmlhttp = new XMLHttpRequest();
var url = "http://alice.com:8080/index.html";

xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {

            // Show the received greeting
            …
        }
}

xmlhttp.open("GET", url, true);
xmlhttp.send();
</script>
</body>
</html>
```

Same Origin Policy – Same Origin request



```
<html>
<body>
<br> Trying to get greetings from myself! </br>
<script>
```

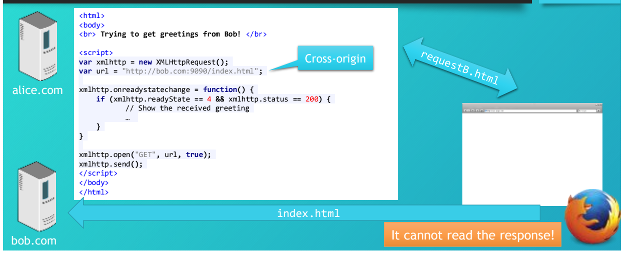
✓	Method	File	Domain	Type	Size	0 ms	80 ms
● 200	GET	requestA.html	alice.com:8080	html	0.73 KB	→ 41 ms	
▲ 304	GET	index.html	alice.com:8080	html	0.18 KB		→ 8 ms

```
xmlhttp.open("GET", url, true);
xmlhttp.send();
</script>
</body>
</html>
```

Demo

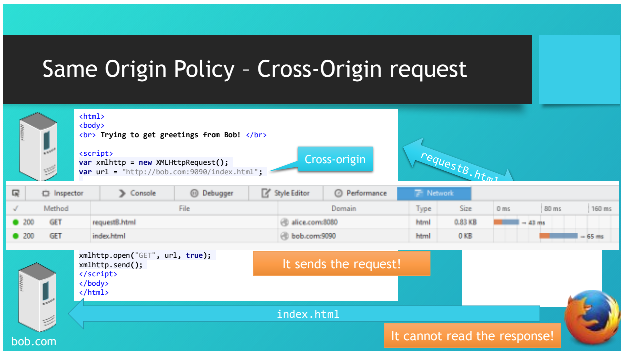
Files: `WebsiteA/requestA.html` and `WebsiteA/index.html`

Same Origin Policy – Cross-Origin request



```
<html>
<body>
<br> Trying to get greetings from Bob! </br>

<script>
var xmlhttp = new XMLHttpRequest();
var url = "http://bob.com:9090/index.html";

xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
            // Show the received greeting
            …
    }
}

xmlhttp.open("GET", url, true);
xmlhttp.send();
</script>
</body>
</html>
```

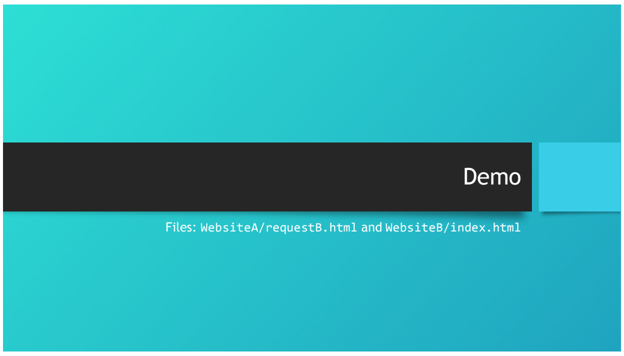
It cannot read the response!



Same Origin Policy – Cross-Origin request

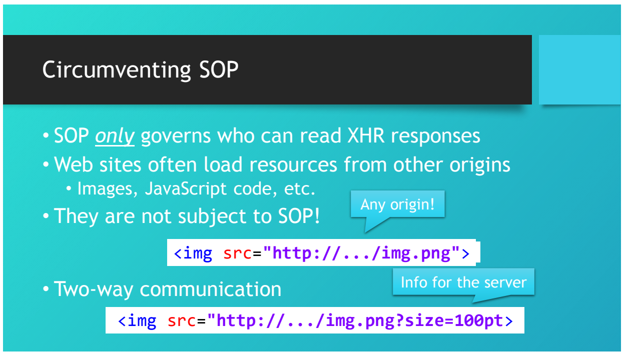
```
<html>
<body>
<br> Trying to get greetings from Bob! </br>

<script>
var xmlhttp = new XMLHttpRequest();
var url = "http://bob.com:9090/index.html";
xmlhttp.open("GET", url, true);
xmlhttp.send();
</script>
</body>
</html>
```

✓	Method	File	Domain	Type	Size	0 ms 80 ms 160 ms
● 200	GET	requestB.html	alice.com:8080	html	0.83 KB	→ 43 ms
● 200	GET	index.html	bob.com:9090	html	0 KB	→ 65 ms



Demo

Files: WebsiteA/requestB.html and WebsiteB/index.html



Circumventing SOP

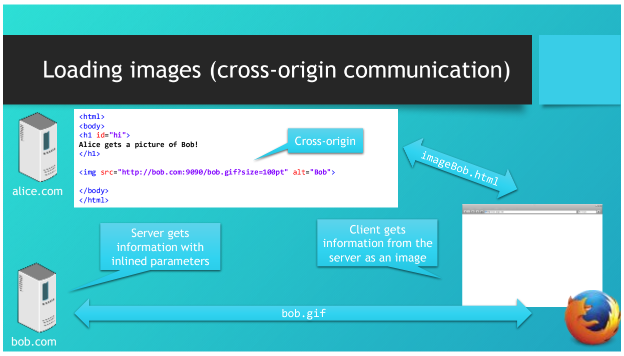
- SOP *only* governs who can read XHR responses
- Web sites often load resources from other origins
  - Images, JavaScript code, etc.
- They are not subject to SOP!

`<img src="http://.../img.png">`

- Two-way communication

`<img src="http://.../img.png?size=100pt">`

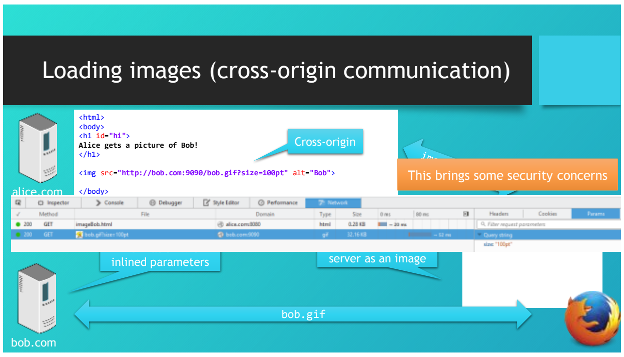
Any origin!
Info for the server



Loading images (cross-origin communication)

```
<html>
<body>
<h1 id="hi">
Alice gets a picture of Bob!
</h1>

<img src="http://bob.com:9090/bob.gif?size=100pt" alt="Bob">

</body>
</html>
```

alice.com
Cross-origin
imageBob.html
Server gets information with inlined parameters
Client gets information from the server as an image
bob.gif
bob.com

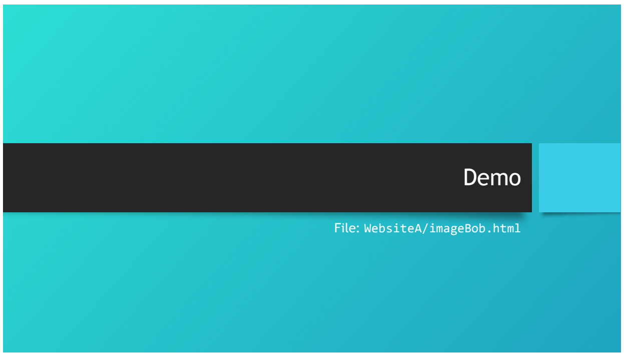


Loading images (cross-origin communication)

```
<html>
<body>
<h1 id="hi">
Alice gets a picture of Bob!
</h1>

<img src="http://bob.com:9090/bob.gif?size=100pt" alt="Bob">

</body>
```

✓	Method	File	Domain	Type	Size	0 ms 80 ms
● 200	GET	imageBob.html	alice.com:8080	html	0.28 KB	→ 20 ms
● 200	GET	bob.gif?size=100pt	bob.com:9090	gif	32.16 KB	→ 52 ms

Query string
size: "100pt"



Demo

File: WebsiteA/imageBob.html

Loading other cross-origin resources

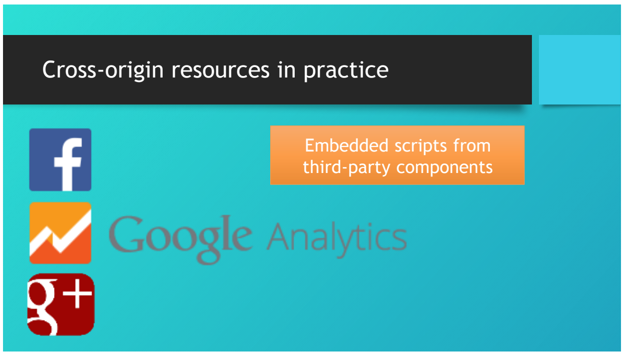
```
<script src="http://.../somecode.js"> </script>
```

Any origin!

- Two-way communication (as for images)
- This brings even more security concerns
  - Fetched code *runs with the same privilege* as the webpage where it gets embedded!

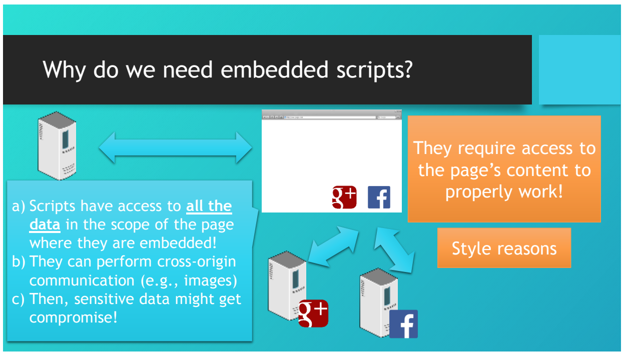
Demo

File: `WebsiteA/jsBob.html`



Cross-origin resources in practice

Embedded scripts from third-party components

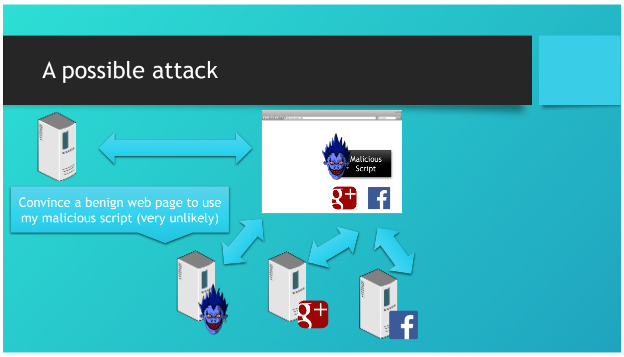


Why do we need embedded scripts?

a) Scripts have access to all the data in the scope of the page where they are embedded!
b) They can perform cross-origin communication (e.g., images)
c) Then, sensitive data might get compromise!

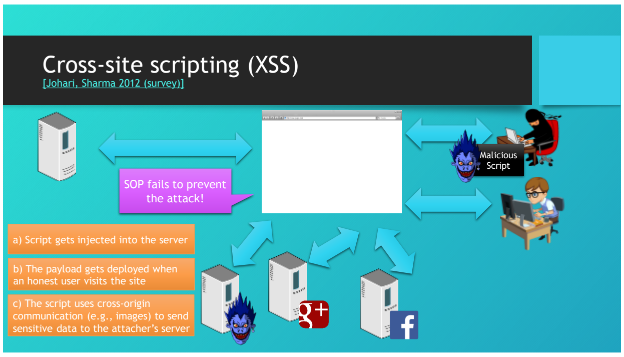
They require access to the page's content to properly work!

Style reasons



A possible attack

Convince a benign web page to use my malicious script (very unlikely)



Cross-site scripting (XSS)

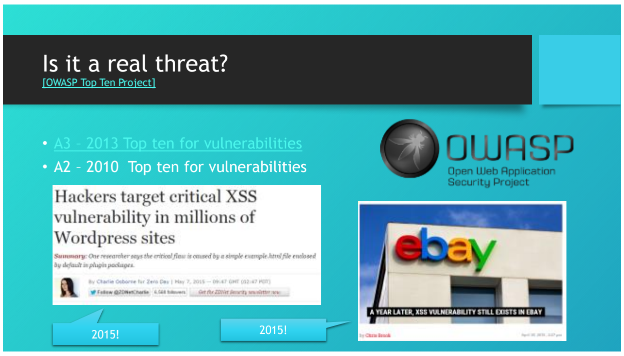
[Johari, Sharma 2012 (survey)]

SOP fails to prevent the attack!

a) Script gets injected into the server

b) The payload gets deployed when an honest user visits the site

c) The script uses cross-origin communication (e.g., images) to send sensitive data to the attacher's server



Is it a real threat?

[OWASP Top Ten Project]

- A3 – 2013 Top ten for vulnerabilities
- A2 – 2010 Top ten for vulnerabilities

Hackers target critical XSS vulnerability in millions of Wordpress sites

Summary: One researcher says the critical flaw is caused by a simple example.html file enclosed by default in plugin packages.

By Charlie Osborne for Zero Day | May 7, 2015 -- 09:47 GMT (02:47 PDT)

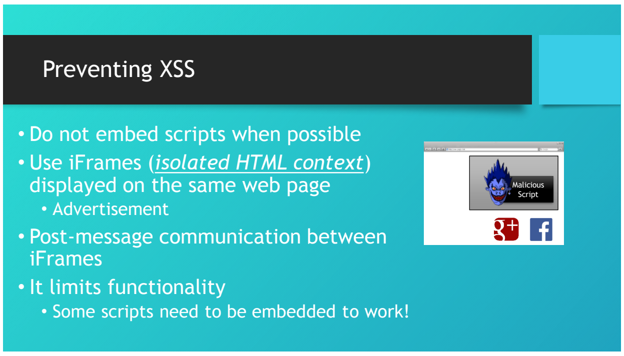
OWASP
Open Web Application Security Project
Follow @ZDNetCharlie
6,568 followers
Get the ZDNet Security newsletter now
ebay

A YEAR LATER, XSS VULNERABILITY STILL EXISTS IN EBAY

by Chris Brook

2015!

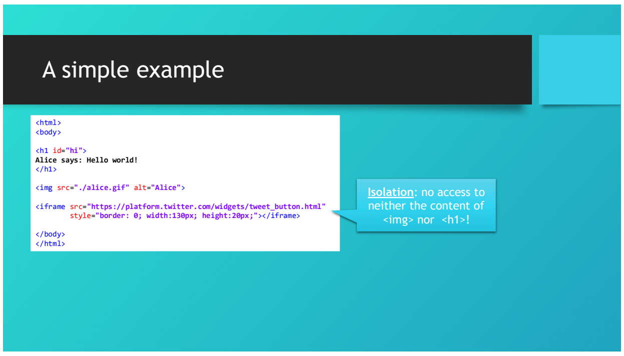
2015!



Preventing XSS

- Do not embed scripts when possible
- Use iFrames (*isolated HTML context*) displayed on the same web page
  - Advertisement
- Post-message communication between iFrames
- It limits functionality
  - Some scripts need to be embedded to work!

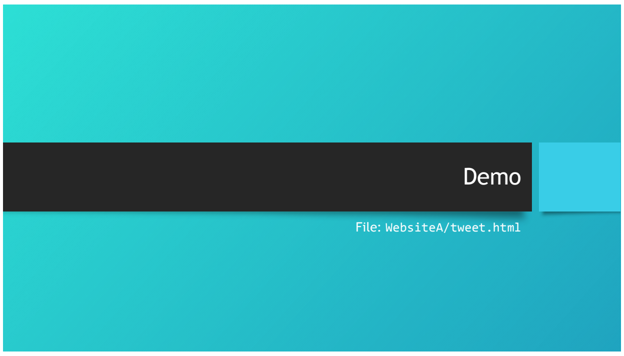
Malicious Script



A simple example

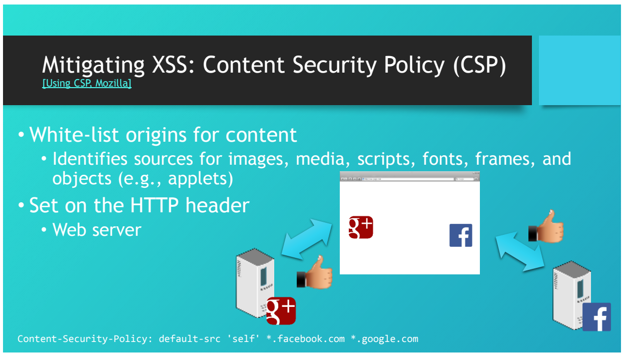
```
<html>
<body>

<h1 id="hi">
Alice says: Hello world!
</h1>

<img src="./alice.gif" alt="Alice">

<iframe src="https://platform.twitter.com/widgets/tweet_button.html"
        style="border: 0; width:130px; height:20px;"></iframe>

</body>
</html>
```

Isolation: no access to neither the content of <img> nor <h1>!



Demo

File: `WebsiteA/tweet.html`

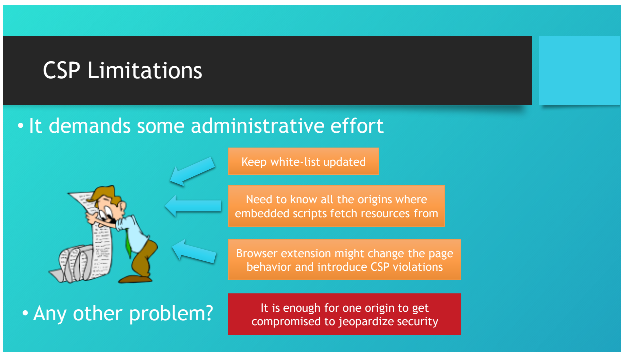


Mitigating XSS: Content Security Policy (CSP)

[Using CSP, Mozilla]

- White-list origins for content
  - Identifies sources for images, media, scripts, fonts, frames, and objects (e.g., applets)
- Set on the HTTP header
  - Web server

```
Content-Security-Policy: default-src 'self' *.facebook.com *.google.com
```



CSP Limitations

- It demands some administrative effort

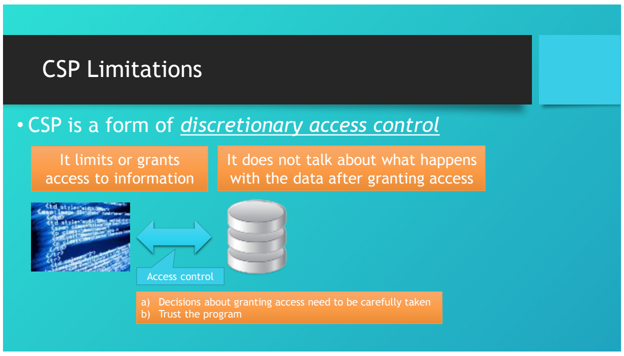
Keep white-list updated

Need to know all the origins where embedded scripts fetch resources from

Browser extension might change the page behavior and introduce CSP violations

- Any other problem?

It is enough for one origin to get compromised to jeopardize security



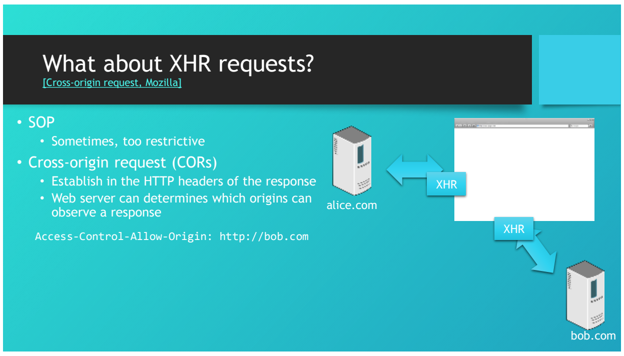
CSP Limitations

- CSP is a form of *discretionary access control*

It limits or grants access to information

It does not talk about what happens with the data after granting access

a) Decisions about granting access need to be carefully taken
b) Trust the program

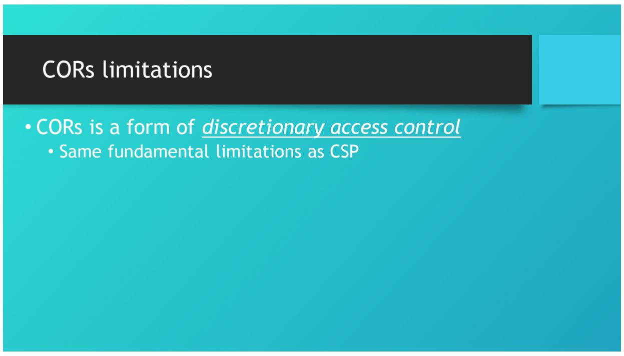


What about XHR requests?

[Cross-origin request, Mozilla]

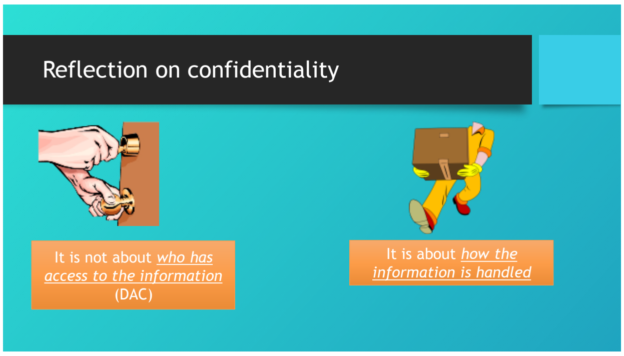
- SOP
  - Sometimes, too restrictive
- Cross-origin request (CORs)
  - Establish in the HTTP headers of the response
  - Web server can determines which origins can observe a response

```
Access-Control-Allow-Origin: http://bob.com
```



CORs limitations

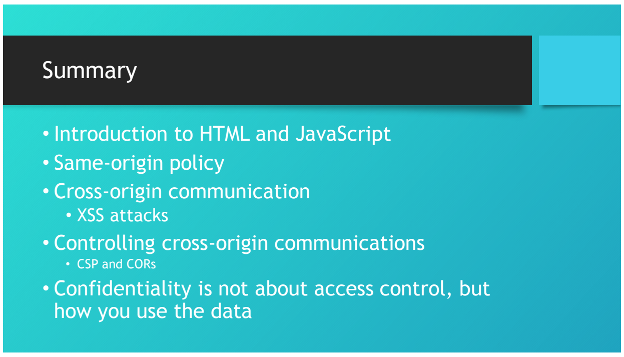
- CORs is a form of *discretionary access control*
  - Same fundamental limitations as CSP



Reflection on confidentiality

It is not about *who has access to the information* (DAC)

It is about *how the information is handled*



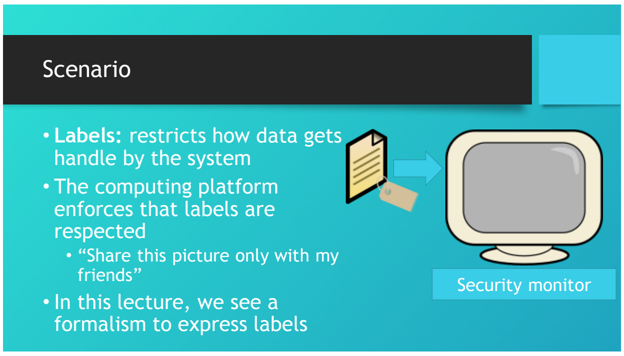
Summary

- Introduction to HTML and JavaScript
- Same-origin policy
- Cross-origin communication
  - XSS attacks
- Controlling cross-origin communications
  - CSP and CORs
- Confidentiality is not about access control, but how you use the data

Security Policies

Disjunction Category Labels

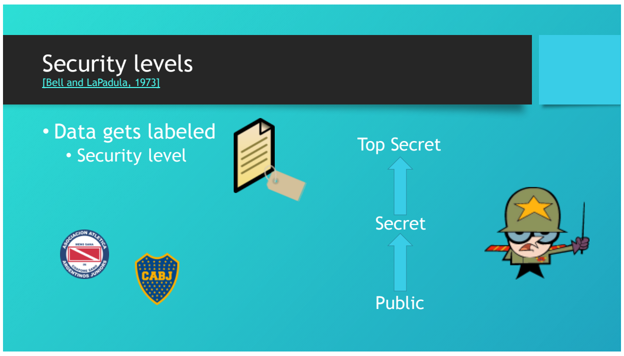
Alejandro Russo
russo@chalmers.se
ECI 2015, UBA, Buenos Aires, Argentina



Scenario

- **Labels:** restricts how data gets handle by the system
- The computing platform enforces that labels are respected
  - "Share this picture only with my friends"
- In this lecture, we see a formalism to express labels

Security monitor

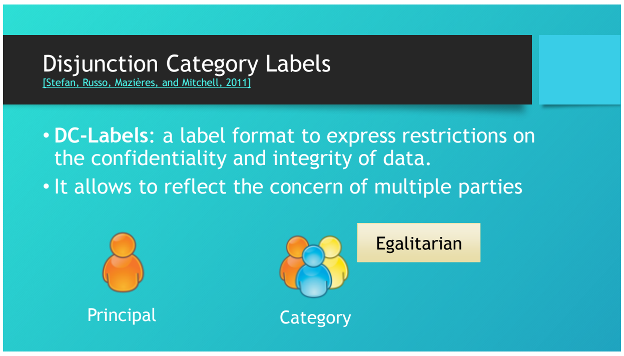


Security levels

[Bell and LaPadula, 1973]

- Data gets labeled
  - Security level

Top Secret
Secret
Public



Disjunction Category Labels

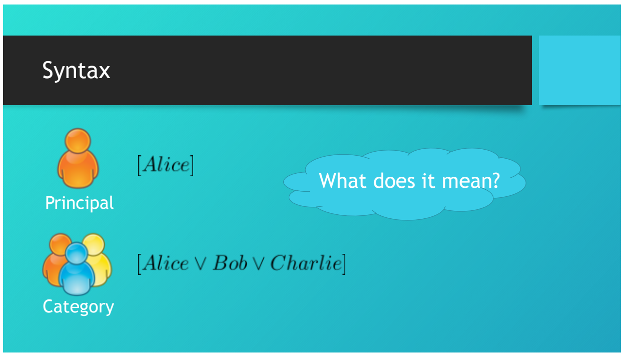
[Stefan, Russo, Mazières, and Mitchell, 2011]

- **DC-Labels:** a label format to express restrictions on the confidentiality and integrity of data.
- It allows to reflect the concern of multiple parties

Principal

Category

Egalitarian

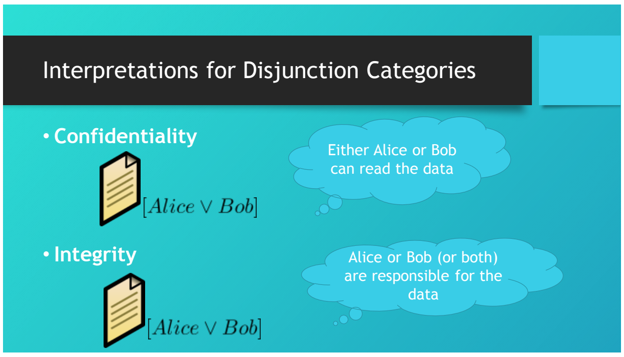


Syntax

Principal $[Alice]$

Category $[Alice \vee Bob \vee Charlie]$

What does it mean?



Interpretations for Disjunction Categories

- **Confidentiality**

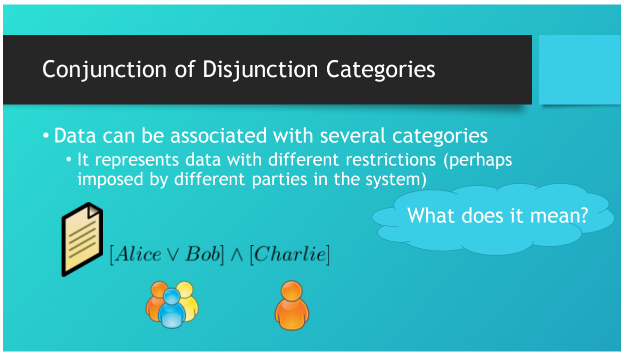
$[Alice \vee Bob]$

Either Alice or Bob can read the data

- **Integrity**

$[Alice \vee Bob]$

Alice or Bob (or both) are responsible for the data

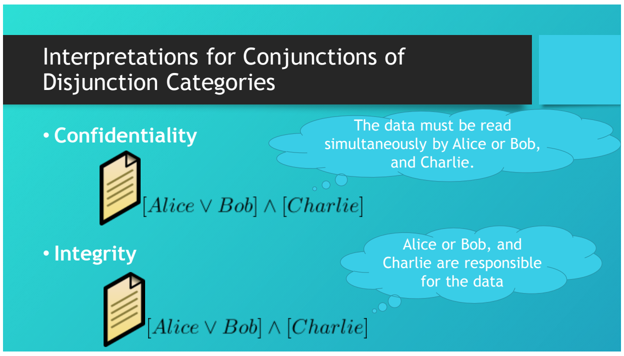


Conjunction of Disjunction Categories

- Data can be associated with several categories
  - It represents data with different restrictions (perhaps imposed by different parties in the system)

What does it mean?

$[Alice \vee Bob] \wedge [Charlie]$



Interpretations for Conjunctions of Disjunction Categories

- **Confidentiality**

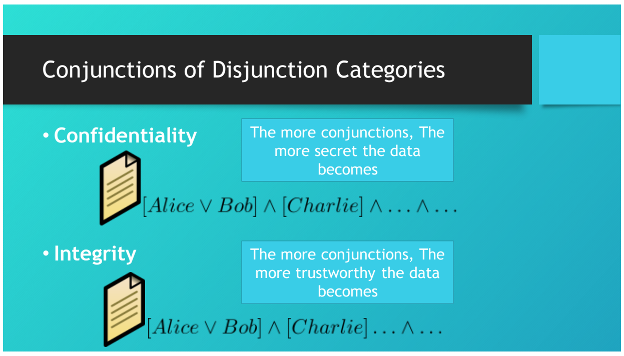
The data must be read simultaneously by Alice or Bob, and Charlie.

$[Alice \vee Bob] \wedge [Charlie]$

- **Integrity**

Alice or Bob, and Charlie are responsible for the data

$[Alice \vee Bob] \wedge [Charlie]$



Conjunctions of Disjunction Categories

- **Confidentiality**

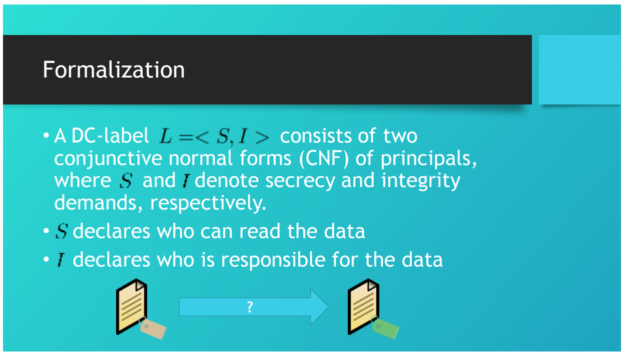
The more conjunctions, The more secret the data becomes

$[Alice \vee Bob] \wedge [Charlie] \wedge \ldots \wedge \ldots$

- **Integrity**

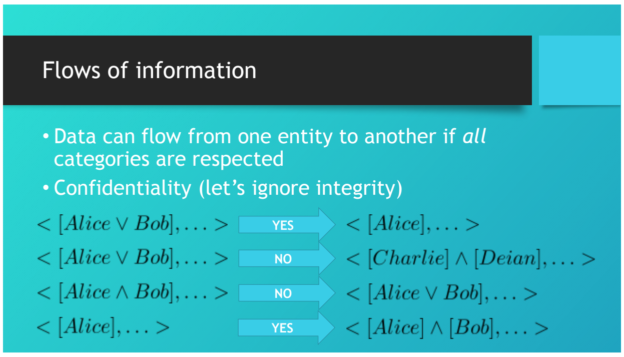
The more conjunctions, The more trustworthy the data becomes

$[Alice \vee Bob] \wedge [Charlie] \ldots \wedge \ldots$



Formalization

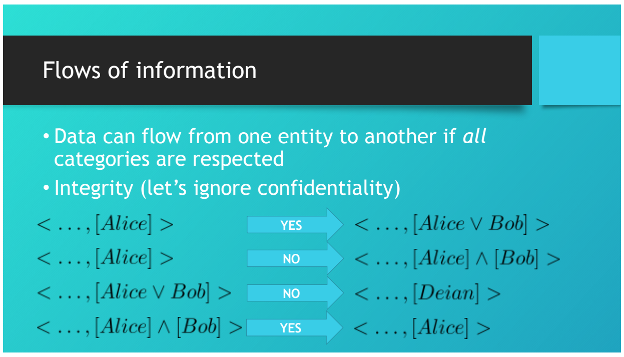
- A DC-label $L = < S, I >$ consists of two conjunctive normal forms (CNF) of principals, where S and I denote secrecy and integrity demands, respectively.
- S declares who can read the data
- I declares who is responsible for the data



Flows of information

- Data can flow from one entity to another if *all* categories are respected
- Confidentiality (let's ignore integrity)

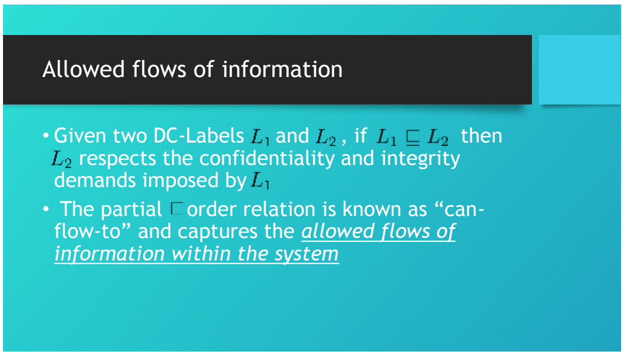
$< [Alice \vee Bob], \ldots >$	YES	$< [Alice], \ldots >$
$< [Alice \vee Bob], \ldots >$	NO	$< [Charlie] \wedge [Deian], \ldots >$
$< [Alice \wedge Bob], \ldots >$	NO	$< [Alice \vee Bob], \ldots >$
$< [Alice], \ldots >$	YES	$< [Alice] \wedge [Bob], \ldots >$



Flows of information

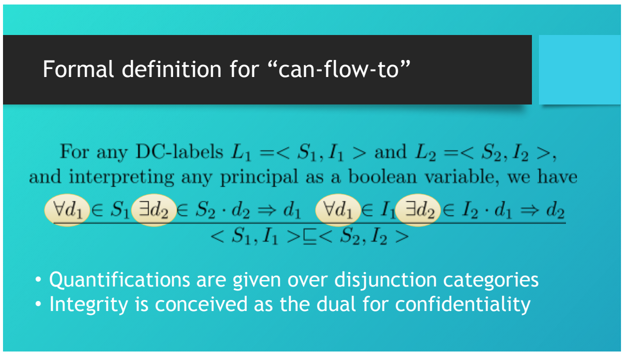
- Data can flow from one entity to another if *all* categories are respected
- Integrity (let's ignore confidentiality)

$< \ldots, [Alice] >$	YES	$< \ldots, [Alice \vee Bob] >$
$< \ldots, [Alice] >$	NO	$< \ldots, [Alice] \wedge [Bob] >$
$< \ldots, [Alice \vee Bob] >$	NO	$< \ldots, [Deian] >$
$< \ldots, [Alice] \wedge [Bob] >$	YES	$< \ldots, [Alice] >$



Allowed flows of information

- Given two DC-Labels L_1 and L_2, if $L_1 \sqsubseteq L_2$ then L_2 respects the confidentiality and integrity demands imposed by L_1
- The partial $\sqsubseteq$ order relation is known as "can-flow-to" and captures the ***allowed flows of information within the system***

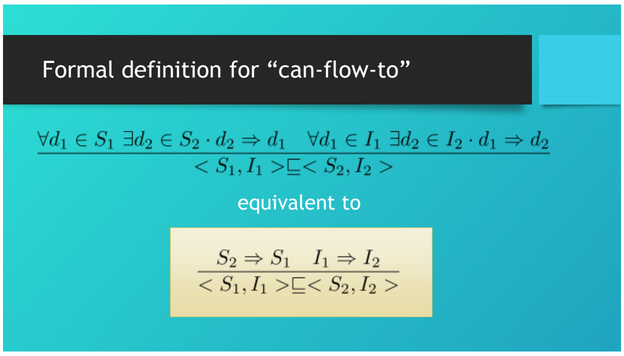


Formal definition for "can-flow-to"

For any DC-labels $L_1 =< S_1, I_1 >$ and $L_2 =< S_2, I_2 >$, and interpreting any principal as a boolean variable, we have

$$\frac{\forall d_1 \in S_1\, \exists d_2 \in S_2 \cdot d_2 \Rightarrow d_1 \quad \forall d_1 \in I_1\, \exists d_2 \in I_2 \cdot d_1 \Rightarrow d_2}{< S_1, I_1 > \sqsubseteq < S_2, I_2 >}$$

- Quantifications are given over disjunction categories
- Integrity is conceived as the dual for confidentiality

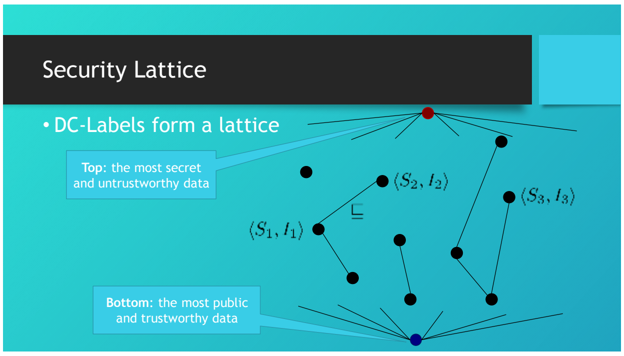


Formal definition for "can-flow-to"

$$\frac{\forall d_1 \in S_1\, \exists d_2 \in S_2 \cdot d_2 \Rightarrow d_1 \quad \forall d_1 \in I_1\, \exists d_2 \in I_2 \cdot d_1 \Rightarrow d_2}{< S_1, I_1 > \sqsubseteq < S_2, I_2 >}$$

equivalent to

$$\frac{S_2 \Rightarrow S_1 \quad I_1 \Rightarrow I_2}{< S_1, I_1 > \sqsubseteq < S_2, I_2 >}$$



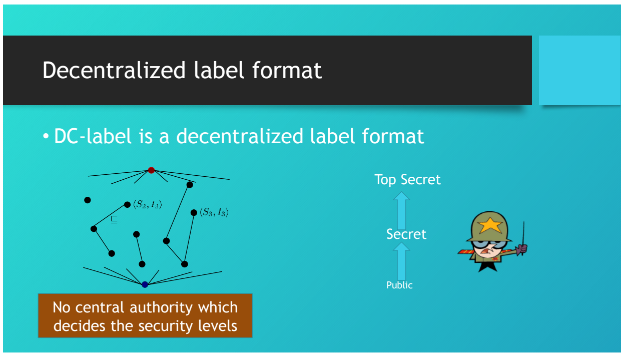
Security Lattice

- DC-Labels form a lattice

$\langle S_2, I_2 \rangle$
$\sqsubseteq$
$\langle S_1, I_1 \rangle$
$\langle S_3, I_3 \rangle$

Top: the most secret and untrustworthy data

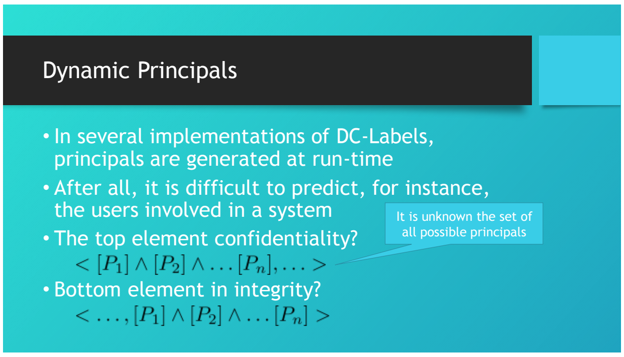
Bottom: the most public and trustworthy data



Decentralized label format

- DC-label is a decentralized label format

No central authority which decides the security levels



Dynamic Principals

- In several implementations of DC-Labels, principals are generated at run-time
- After all, it is difficult to predict, for instance, the users involved in a system
- The top element confidentiality?
  $< [P_1] \wedge [P_2] \wedge \ldots [P_n], \ldots >$
- Bottom element in integrity?
  $< \ldots, [P_1] \wedge [P_2] \wedge \ldots [P_n] >$

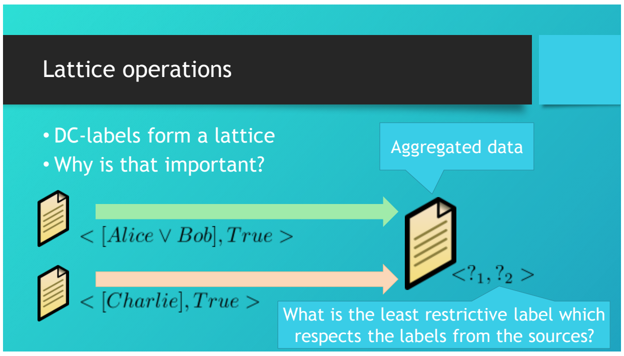
It is unknown the set of all possible principals

Top and bottom elements

$$\frac{S_2 \Rightarrow S_1 \quad I_1 \Rightarrow I_2}{< S_1, I_1 > \sqsubseteq < S_2, I_2 >}$$

$\bot \sqsubseteq < S, I >$ → $\bot = < True, False >$

$< S, I > \sqsubseteq \top$ → $\top = < False, True >$



Lattice operations

- DC-labels form a lattice
- Why is that important?

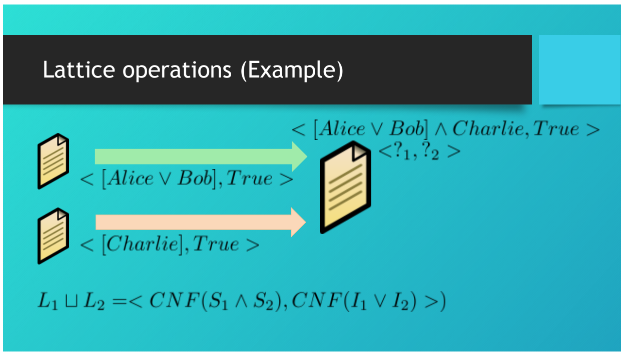
What is the least restrictive label which respects the labels from the sources?

Join and meet operations

For any DC-labels $L_1 =< S_1, I_1 >$ and $L_2 =< S_2, I_2 >$, the join ($\sqcup$) and meet ($\sqcap$) operations are defined as:

$$L_1 \sqcup L_2 =< CNF(S_1 \wedge S_2), CNF(I_1 \vee I_2) >)$$
$$L_1 \sqcap L_2 =< CNF(S_1 \vee S_2), CNF(I_1 \wedge I_2) >)$$

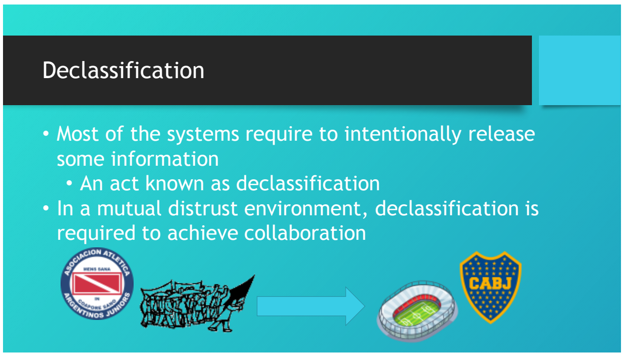
- Conjunction and disjunction of DC-labels' components might introduce redundant categories.
- Apply a Conjunctive Normal Form (CNF) reduction



Lattice operations (Example)

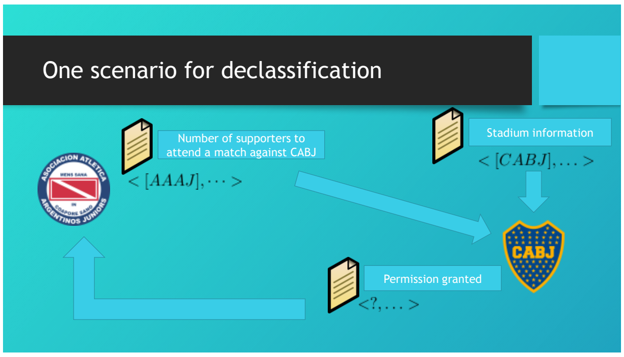
$< [Alice \vee Bob] \wedge Charlie, True >$
$<?_1, ?_2 >$
$< [Alice \vee Bob], True >$
$< [Charlie], True >$

$$L_1 \sqcup L_2 =< CNF(S_1 \wedge S_2), CNF(I_1 \vee I_2) >)$$

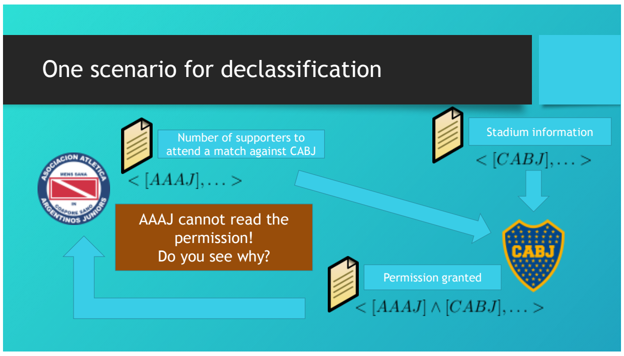


Declassification

- Most of the systems require to intentionally release some information
  - An act known as declassification
- In a mutual distrust environment, declassification is required to achieve collaboration



One scenario for declassification



One scenario for declassification

Number of supporters to attend a match against CABJ

$< [AAAJ], \ldots >$

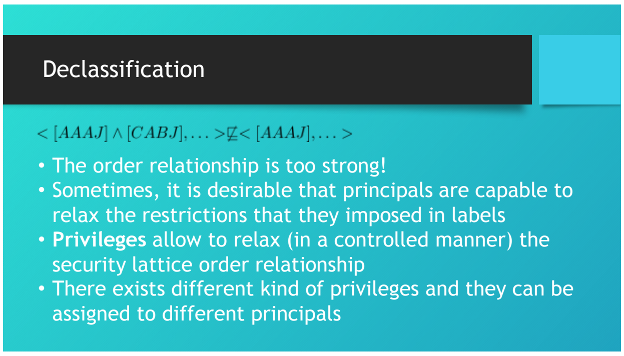
Stadium information

$< [CABJ], \ldots >$

AAAJ cannot read the permission!
Do you see why?

Permission granted

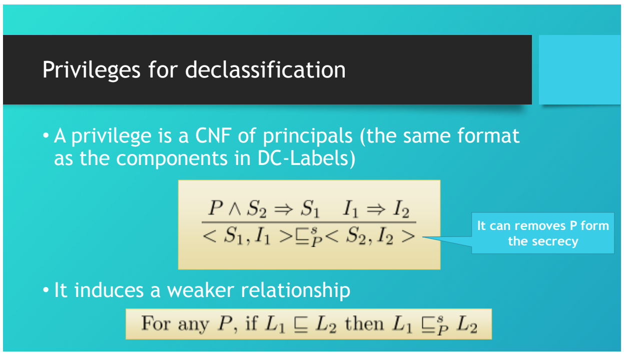
$< [AAAJ] \wedge [CABJ], \ldots >$



Declassification

$< [AAAJ] \wedge [CABJ], \ldots > \not\sqsubseteq < [AAAJ], \ldots >$

- The order relationship is too strong!
- Sometimes, it is desirable that principals are capable to relax the restrictions that they imposed in labels
- **Privileges** allow to relax (in a controlled manner) the security lattice order relationship
- There exists different kind of privileges and they can be assigned to different principals



Privileges for declassification

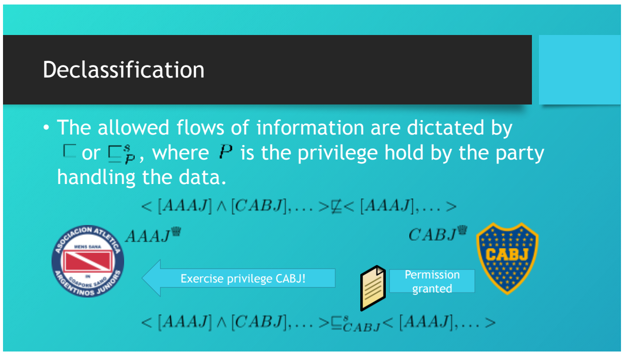
- A privilege is a CNF of principals (the same format as the components in DC-Labels)

$$\frac{P \wedge S_2 \Rightarrow S_1 \quad I_1 \Rightarrow I_2}{< S_1, I_1 > \sqsubseteq^s_P < S_2, I_2 >}$$

It can removes P form the secrecy

- It induces a weaker relationship

For any P, if $L_1 \sqsubseteq L_2$ then $L_1 \sqsubseteq^s_P L_2$



Declassification

- The allowed flows of information are dictated by $\sqsubseteq$ or $\sqsubseteq^s_P$, where P is the privilege hold by the party handling the data.

$< [AAAJ] \wedge [CABJ], \ldots > \not\sqsubseteq < [AAAJ], \ldots >$

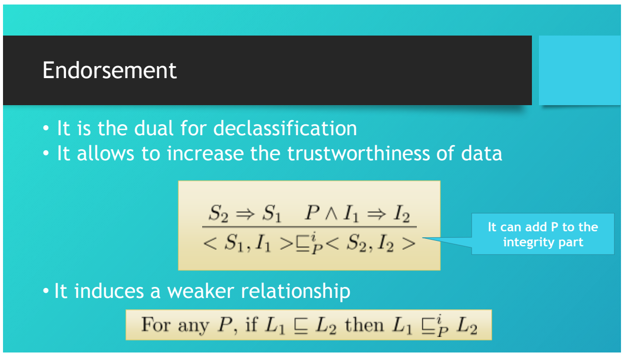
$AAAJ$♛

$CABJ$♛

Exercise privilege CABJ!

Permission granted

$< [AAAJ] \wedge [CABJ], \ldots > \sqsubseteq^s_{CABJ} < [AAAJ], \ldots >$



Endorsement

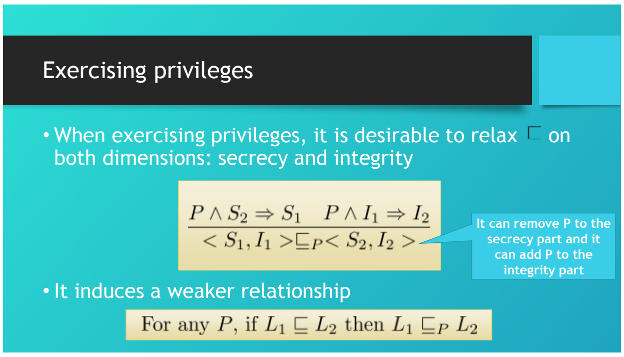
- It is the dual for declassification
- It allows to increase the trustworthiness of data

$$\frac{S_2 \Rightarrow S_1 \quad P \wedge I_1 \Rightarrow I_2}{< S_1, I_1 > \sqsubseteq^i_P < S_2, I_2 >}$$

It can add P to the integrity part

- It induces a weaker relationship

For any P, if $L_1 \sqsubseteq L_2$ then $L_1 \sqsubseteq^i_P L_2$



Exercising privileges

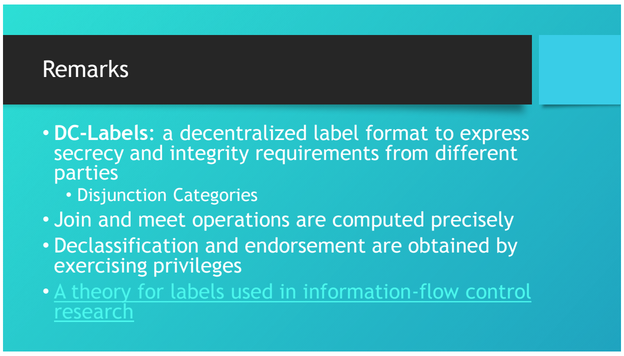
- When exercising privileges, it is desirable to relax $\sqsubseteq$ on both dimensions: secrecy and integrity

$$\frac{P \wedge S_2 \Rightarrow S_1 \quad P \wedge I_1 \Rightarrow I_2}{< S_1, I_1 > \sqsubseteq_P < S_2, I_2 >}$$

It can remove P to the secrecy part and it can add P to the integrity part

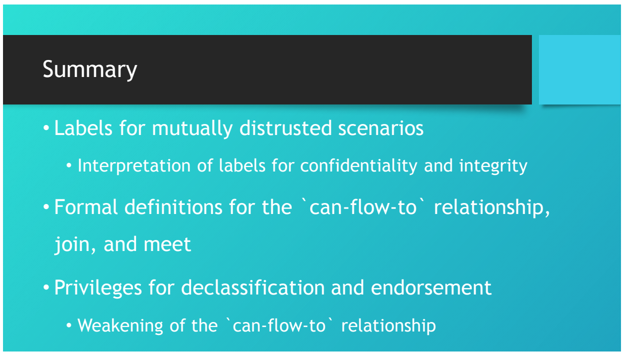
- It induces a weaker relationship

For any P, if $L_1 \sqsubseteq L_2$ then $L_1 \sqsubseteq_P L_2$



Remarks

- **DC-Labels:** a decentralized label format to express secrecy and integrity requirements from different parties
  - Disjunction Categories
- Join and meet operations are computed precisely
- Declassification and endorsement are obtained by exercising privileges
- A theory for labels used in information-flow control research

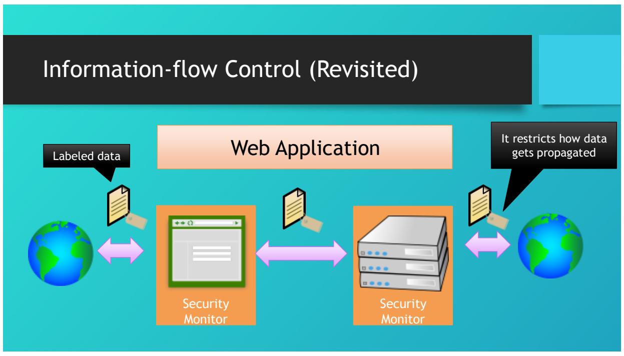


Summary

- Labels for mutually distrusted scenarios
  - Interpretation of labels for confidentiality and integrity
- Formal definitions for the `can-flow-to` relationship, join, and meet
- Privileges for declassification and endorsement
  - Weakening of the `can-flow-to` relationship

Security Labels For The Web

Alejandro Russo
russo@chalmers.se
ECI 2015, UBA, Buenos Aires, Argentina

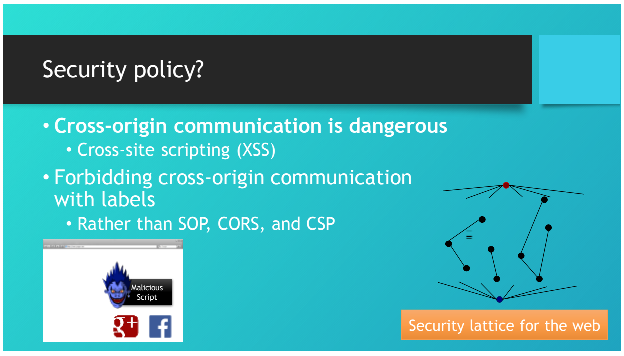


Information-flow Control (Revisited)

Labeled data

Web Application

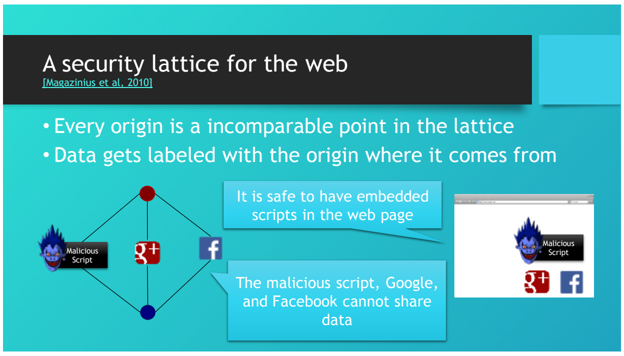
It restricts how data gets propagated



Security policy?

- **Cross-origin communication is dangerous**
  - Cross-site scripting (XSS)
- Forbidding cross-origin communication with labels
  - Rather than SOP, CORS, and CSP

Security lattice for the web



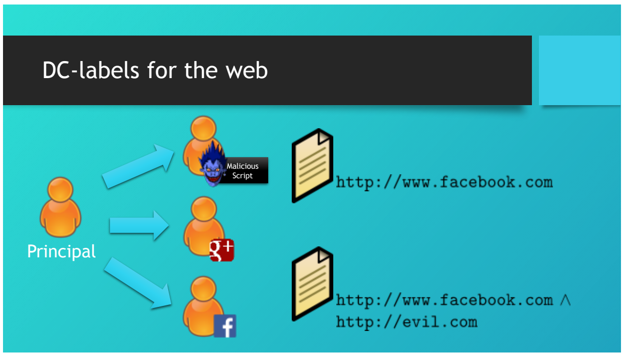
A security lattice for the web

[Magazinius et al. 2010]

- Every origin is a incomparable point in the lattice
- Data gets labeled with the origin where it comes from

It is safe to have embedded scripts in the web page

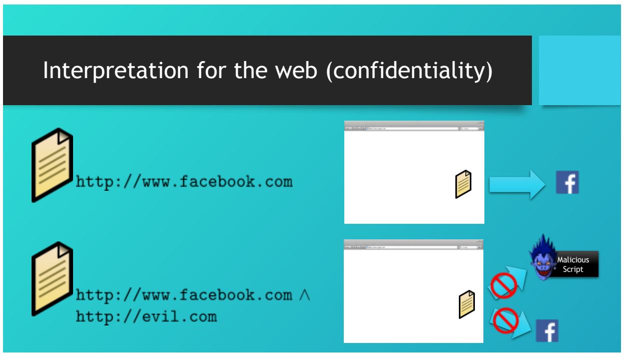
The malicious script, Google, and Facebook cannot share data



DC-labels for the web

```
http://www.facebook.com
```

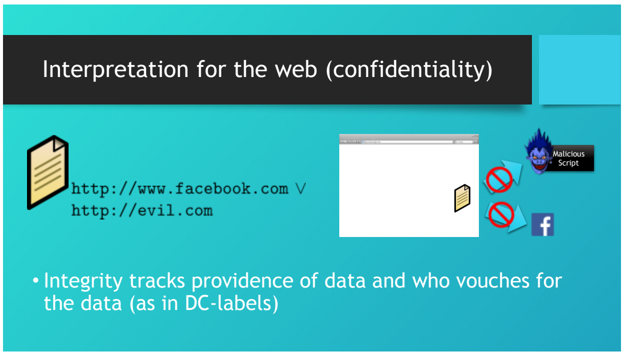
```
http://www.facebook.com ∧
http://evil.com
```



Interpretation for the web (confidentiality)

```
http://www.facebook.com
```

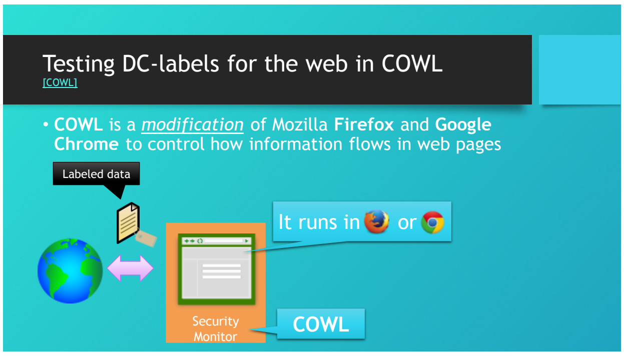
```
http://www.facebook.com ∧
http://evil.com
```



Interpretation for the web (confidentiality)

http://www.facebook.com ∨
http://evil.com

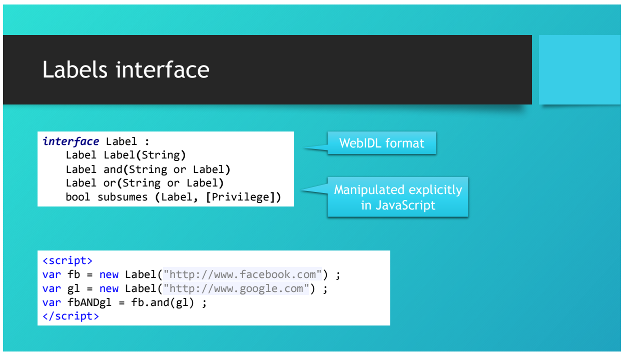
- Integrity tracks providence of data and who vouches for the data (as in DC-labels)



Testing DC-labels for the web in COWL

[COWL]

- **COWL** is a *modification* of Mozilla **Firefox** and **Google Chrome** to control how information flows in web pages

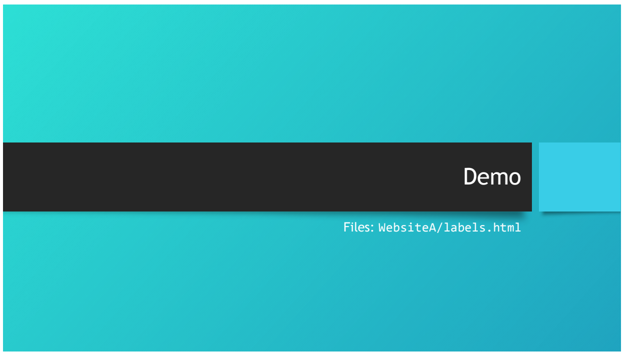


Labels interface

```
interface Label :
    Label Label(String)
    Label and(String or Label)
    Label or(String or Label)
    bool subsumes (Label, [Privilege])
```

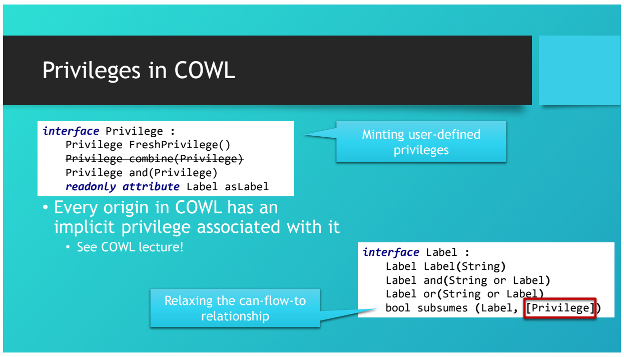
WebIDL format
Manipulated explicitly in JavaScript

```
<script>
var fb = new Label("http://www.facebook.com") ;
var gl = new Label("http://www.google.com") ;
var fbANDgl = fb.and(gl) ;
</script>
```



Demo

Files: WebsiteA/labels.html



Privileges in COWL

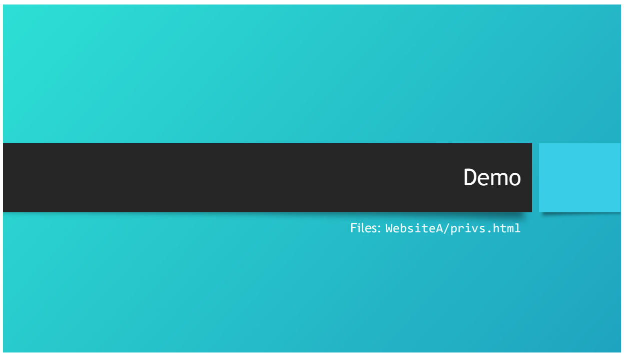
```
interface Privilege :
    Privilege FreshPrivilege()
    ~~Privilege combine(Privilege)~~
    Privilege and(Privilege)
    readonly attribute Label asLabel
```

Minting user-defined privileges

- Every origin in COWL has an implicit privilege associated with it
    - See COWL lecture!

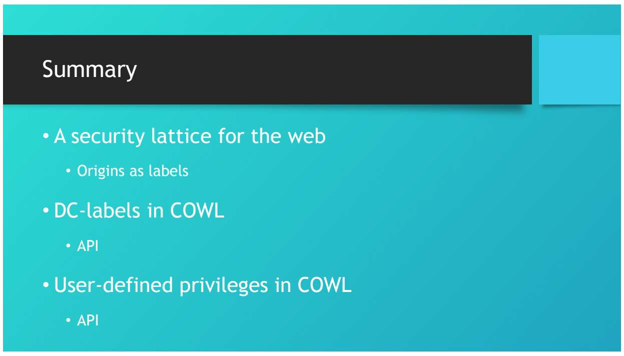
```
interface Label :
    Label Label(String)
    Label and(String or Label)
    Label or(String or Label)
    bool subsumes (Label, [Privilege])
```

Relaxing the can-flow-to relationship



Demo

Files: WebsiteA/privs.html



Summary

- A security lattice for the web
  - Origins as labels
- DC-labels in COWL
  - API
- User-defined privileges in COWL
  - API

Disjunction Category Labels

Deian Stefan[1], Alejandro Russo[2], David Mazières[1], and John C. Mitchell[1]

[1] Stanford University
[2] Chalmers University of Technology

Abstract. We present disjunction category (DC) labels, a new label format for enforcing information flow in the presence of mutually distrusting parties. DC labels can be ordered to form a lattice, based on propositional logic implication and conjunctive normal form. We introduce and prove soundness of decentralized privileges that are used in declassifying data, in addition to providing a notion of privilege-hierarchy. Our model is simpler than previous decentralized information flow control (DIFC) systems and does not rely on a centralized principal hierarchy. Additionally, DC labels can be used to enforce information flow both statically and dynamically. To demonstrate their use, we describe two Haskell implementations, a library used to perform dynamic label checks, compatible with existing DIFC systems, and a prototype library that enforces information flow statically, by leveraging the Haskell type checker.

Keywords: Security, labels, decentralized information flow control, logic

1 Introduction

Information flow control (IFC) is a general method that allows components of a system to be passed sensitive information and restricts its use in each component. Information flow control can be used to achieve confidentiality, by preventing unwanted information leakage, and integrity, by preventing unreliable information from flowing into critical operations. Modern IFC systems typically label data and track labels, while allowing users exercising appropriate privileges to explicitly downgrade information themselves. While the IFC system cannot guarantee that downgrading preserves the desired information flow properties, it is possible to identify all the downgrading operations and limit code audit to these portions of the code. Overall, information flow systems make it possible to build applications that enforce end-to-end security policies even in the presence of untrusted code.

We present disjunction category (DC) labels: a new label format for enforcing information flow in systems with mutually distrusting parties. By formulating DC labels using propositional logic, we make it straightforward to verify conventional lattice conditions and other useful properties. We introduce and prove soundness of decentralized privileges that are used in declassifying data, and provide a notion of privilege-hierarchy. Compared to Myers and Liskov's decentralized label model (DLM) [21], for example, our model is simpler and does not rely on a centralized principal hierarchy. Additionally, DC labels can be used to enforce information flow both statically and dynamically, as shown in our Haskell implementations.

A DC label, written $\langle S, I\rangle$, consists of two Boolean formulas over principals, the first specifying secrecy requirements and the second specifying integrity requirements. Information flow is restricted according to implication of these formulas in a way that preserves secrecy and integrity. Specifically, secrecy of information labeled $\langle S, I\rangle$ is preserved by requiring that a receiving channel have a stronger secrecy requirement S' that *implies* S, while integrity requires the receiver to have a weaker integrity requirement I' that is *implied by* I. These two requirements are combined to form a can-flow-to relation, which provides a partial order on the set of DC labels that also has the lattice operations meet and join.

Our decentralized privileges can be delegated in a way that we prove preserves confidentiality and integrity properties, resulting in a privilege hierarchy. Unlike [21], this is accomplished without a notion of "can act for" or a central principal hierarchy. Although our model can be extended to support revocation using approaches associated with public key infrastructures, we present a potentially more appealing selective revocation approach that is similar to those used in capability-based systems.

We illustrate the expressiveness of DC labels by showing how to express several common design patterns. These patterns are based in part on security patterns used in capability-based systems. Confinement is achieved by labeling data so that it cannot be read and exfiltrated to the network by arbitrary principals. A more subtle pattern that relies on the notion of clearance is used to show how a process can be restricted from even accessing overly-sensitive information (e.g., private keys); this pattern is especially useful when covert channels are a concern. We also describe privilege separation and user authentication patterns. As described more fully later in the paper, privilege separation may be achieved using delegation to subdivide the privileges of a program and compartmentalize a program into components running with fewer privileges. The user authentication pattern shows how to leverage a login client that users trust with their username and password (since the user supplies them as input), without unnecessarily creating other risks.

We describe two Haskell implementations: a library used to perform dynamic label checks, compatible with existing DIFC systems, and a prototype library that enforces information flow statically by leveraging Haskell's type checker.

The remainder of the paper is structured as follows. In Section 2, we introduce DC labels and present some of their properties. Section 3 presents semantics and soundness proofs for our DC label system. Design patterns are presented and explained in Section 5, with the implementations presented in Section 6. We summarize related work in Section 7 and conclude in Section 8.

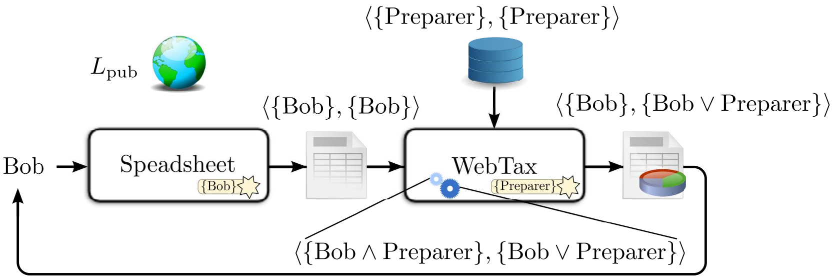


Fig. 1: A tax preparation system with mutually distrusting parties.

2 DC Label Model

In a DIFC system, every piece of data is *labeled*, or "tagged." Labels provide a means for tracking, and, more importantly, controlling the propagation of information according to a security policy, such as *non-interference* [10].

DC labels can be used to express a conjunction of restrictions on information flow that represents the interests of multiple stake-holders. As a result, DC labels are especially suitable for systems in which participating parties do not fully trust each other. Fig. 1 presents an example, originally given in [21], that illustrates such a system. Here, user Bob firstly inputs his tax information into the *Spreadsheet* program, which he fully trusts. The data is then exported to another program, called *WebTax*, for final analysis. Though conceptually simple, several challenges arise since Bob does not trust WebTax with his data. Without inspecting WebTax, Bob cannot be sure that his privacy policies are respected and his tax information is not exfiltrated to the network. Analogously, the WebTax author, called Preparer, does not entrust Bob with the source code. Furthermore, the tax preparation program relies on a proprietary database and Preparer wishes to assert that even if the program contains bugs, the proprietary database information cannot be leaked to the public network. It is clear that even for such a simple example the end-to-end guarantees are difficult to satisfy with more-traditional access control mechanisms. Using IFC, however, these security policies can be expressed naturally and with minimal trust. Specifically, the parties only need to trust the system IFC-enforcement mechanism; programs, including WebTax, can be executed with no implicit trust. We now specify DC labels and show their use in enforcing the policies of this example.

As previously mentioned, a DC label consists of two Boolean formulas over principals. We make a few restrictions on the labels' format in order to obtain a unique representation for each label and an efficient and decidable can-flow-to relationship.

Definition 1 (DC Labels). *A* DC label, *written* $\langle S, I\rangle$, *is a pair of Boolean formulas over principals such that:*

- *Both S and I are minimal formulas in conjunctive normal form (CNF), with terms and clauses sorted to give each formula a unique representation, and*

– *Neither S nor I contains any negated terms.*

In a DC label, S protects secrecy by specifying the principals that are allowed (or whose consent is needed) to observe the data. Dually, I protects integrity by specifying principals who created and may currently modify the data. For example, in the system of Fig. 1, Bob and Preparer respectively label their data $\langle \{\text{Bob}\}, \{\text{Bob}\} \rangle$ and $\langle \{\text{Preparer}\}, \{\text{Preparer}\} \rangle$, specifying that they created the data and they are the only observers.

Data may flow between differently labeled entities, but only in such a way as to accumulate additional secrecy restrictions or be stripped in integrity ones, not vice versa. Specifically there is a partial order, written $\sqsubseteq$ ("can-flow-to"), that specifies when data can flow between labeled entities. We define $\sqsubseteq$ based on logical implication ($\Longrightarrow$) as follows:

Definition 2 (can-flow-to relation). *Given any two DC labels $L_1 = \langle S_1, I_1 \rangle$ and $L_2 = \langle S_2, I_2 \rangle$, the can-flow-to relation is defined as:*

$$\frac{S_2 \implies S_1 \qquad I_1 \implies I_2}{\langle S_1, I_1 \rangle \sqsubseteq \langle S_2, I_2 \rangle}$$

In other words, data labeled $\langle S_1, I_1 \rangle$ can flow into an entity labeled $\langle S_2, I_2 \rangle$ as long as the secrecy of the data, and integrity of the entity are preserved. Intuitively, the $\sqsubseteq$ relation imposes the restriction that any set of principals who can observe data afterwards must also have been able to observe it earlier. For instance, it is permissible to have $S_2 = \{\text{Bob} \wedge \text{Preparer}\}$ and $S_1 = \text{Bob}$, because $S_2 \implies S_1$, and Bob's consent is still required to observe data with the new label. Dually, integrity of the entity is preserved by requiring that the source label impose more restrictions than that of the destination.

In our model, public entities (e.g., network interface in Fig. 1) have the default, or *empty* label, $\langle \textbf{True}, \textbf{True} \rangle$, written L_{pub}. Although specified by the label $\langle S, I \rangle$, it is intuitive that data labeled as such can be written to a public network with label L_{pub}, only with the permission of a set of principals satisfying the Boolean formula S. Conversely, data read from the network can be labeled $\langle S, I \rangle$ only with the permission of a set of principals satisfying I.

In an IFC system, label checks using the can-flow-to relation are performed at every point of possible information flow. Thus, if the WebTax program of Fig. 1 attempts to write Bob or Preparer's data to the network interface, either by error or malfeasance, both label checks $\langle \{\text{Bob}\}, \{\text{Bob}\} \rangle \sqsubseteq L_{\text{pub}}$ and $\langle \{\text{Preparer}\}, \{\text{Preparer}\} \rangle \sqsubseteq L_{\text{pub}}$ will fail. However, the system must also label the intermediate results of a WebTax computation (on Bob and Preparer's joint data) such that they can only be observed and written to the network if both principals consent.

The latter labeling requirement is recurring and directly addressed by a core property of many IFC systems: the label lattice property [4]. Specifically, for any two labels L_1, L_2 the lattice property states that there is a well defined, *least upper bound* (*join*), written $L_1 \sqcup L_2$, and *greatest lower bound* (*meet*), written

$L_1 \sqcap L_2$, such that $L_i \sqsubseteq L_1 \sqcup L_2$ and $L_1 \sqcap L_2 \sqsubseteq L_i$ for L_i and $i = 1, 2$. We define the join and meet for DC labels as follows.

Definition 3 (Join and meet for DC labels). *The join and meet of any two DC labels* $L_1 = \langle S_1, I_1 \rangle$ *and* $L_2 = \langle S_2, I_2 \rangle$ *are respectively defined as:*

$$L_1 \sqcup L_2 = \langle S_1 \wedge S_2, I_1 \vee I_2 \rangle$$
$$L_1 \sqcap L_2 = \langle S_1 \vee S_2, I_1 \wedge I_2 \rangle$$

where each component of the resulting labels is reduced to CNF.

Intuitively, the secrecy component of the join protects the secrecy of L_1 and L_2 by specifying that both set of principals, those appearing in S_1 and those in S_2, must consent for data labeled $S_1 \wedge S_2$ to be observed. Conversely, the integrity component of the join, $I_1 \vee I_2$, specifies that either principals of I_1 or I_2 could have created and modify the data. Dual properties hold for the meet $L_1 \sqcap L_2$, a label computation necessary when labeling an object that is written to multiple entities. We note that although we use $I_1 \vee I_2$ informally, by definition, a DC label component must be in CNF. Reducing logic formulas, such as $I_1 \vee I_2$, to CNF is standard [23], and we do not discuss it further.

Revisiting the example of Fig. 1, we highlight that the intermediate results generated by the WebTax program from both Bob and Preparer's data are labeled by the join $\langle \{\text{Bob}\}, \{\text{Bob}\} \rangle \sqcup \langle \{\text{Preparer}\}, \{\text{Preparer}\} \rangle$ which is reduced to $\langle \{\text{Bob} \wedge \text{Preparer}\}, \{\text{Bob} \vee \text{Preparer}\} \rangle$. The secrecy component of the label confirms our intuition that the intermediate results are composed of both party's data and thus the consent of both Bob and Preparer is needed to observe it. In parallel, the integrity component agrees with the intuition that the intermediate results could have been created from Bob or Preparer's data.

2.1 Declassification and endorsement

We model both declassification and endorsement as principals explicitly deciding to exercise *privileges*. When code exercises privileges, it means code acting on behalf of a combination of principals is requesting an action that might violate the can-flow-to relation. For instance, if the secrecy component of a label is $\{\text{Bob} \wedge \text{Preparer}\}$, then by definition code must act on behalf of both Bob and the Preparer to transmit the data over a public network. However, what if the Preparer unilaterally wishes to change the secrecy label on data from $\{\text{Bob} \wedge \text{Preparer}\}$ to $\{\text{Bob}\}$ (as to release the results to Bob)? Intuitively, such a partial declassification should be allowed, because the data still cannot be transmitted over the network without Bob's consent. Hence, if the data is eventually made public, both Bob and the Preparer will have consented, even if not simultaneously.

We formalize such partial declassification by defining a more permissive pre-order, $\sqsubseteq_P$ ("can-flow-to given privileges P"). $L_1 \sqsubseteq_P L_2$ means that when exercising privileges P, it is permissible for data to flow from an entity labeled L_1

to one labeled L_2. $L_1 \sqsubseteq L_2$ trivially implies $L_1 \sqsubseteq_P L_2$ for any privileges P, but for non-empty P, there exist labels for which $L_1 \sqsubseteq_P L_2$ even though $L_1 \not\sqsubseteq L_2$.

We represent privileges P as a conjunction of principals for whom code is acting. (Actually, P can be a more general Boolean formula like label components, but the most straight-forward use is as a simple conjunction of principals.) We define $\sqsubseteq_P$ as follows:

Definition 4 (can-flow-to given privileges relation). *Given a Boolean formula P representing privileges and any two DC labels $L_1 = \langle S_1, I_1\rangle$ and $L_2 = \langle S_2, I_2\rangle$, the can-flow-to given privileges P relation is defined as:*

$$\frac{P \wedge S_2 \implies S_1 \qquad P \wedge I_1 \implies I_2}{\langle S_1, I_1\rangle \sqsubseteq_P \langle S_2, I_2\rangle}$$

Recall that without exercising additional privileges, data labeled $\langle S, I\rangle$ can be written to a public network, labeled L_{pub}, only with the permission of a set of principals satisfying the Boolean formula S, while data read from a public network can be labeled $\langle S, I\rangle$ only with the permission of a set of principals satisfying I. Considering additional privileges, it is easy to see that $\langle S, I\rangle \sqsubseteq_P L_{\text{pub}}$ iff $P \implies S$ and, conversely, $L_{\text{pub}} \sqsubseteq_P \langle S, I\rangle$ iff $P \implies I$. In other words, code exercising privileges P can declassify and write data to the public network if P implies the secrecy label of that data, and can similarly incorporate and endorse data from the public network if P implies the integrity label.

In our WebTax example, the Spreadsheet program runs on behalf of Bob and exercises the {Bob} privilege to endorse data sent to WebTax. Conversely, the WebTax program is executed with the {Preparer} privilege which it exercises when declassifying results from {Bob$\wedge$Preparer} to {Bob}; as expected, to allow Bob to observe the results, this declassification step is crucial.

It is a property of our system that exporting data through multiple exercises of privilege cannot reduce the overall privilege required to export data. For instance, if $\langle S, I\rangle \sqsubseteq_{P_1} \langle S', I'\rangle \sqsubseteq_{P_2} L_{\text{pub}}$, it must be that $P_1 \wedge P_2 \implies S$, since $P_2 \implies S'$ and $P_1 \wedge S' \implies S$. A similar, and dual, property holds for multiple endorsements.

The mechanisms provided by $\sqsubseteq_P$ corresponds to the *who* dimension of declassification [25], i.e., whoever has the privileges P can use the relationship $\sqsubseteq_P$ to release (endorse) information. With minimal encoding, it is also possible to address the *what* and *when* dimension using $\sqsubseteq_P$. Specifically, the what dimension can be addressed by carefully designing the data type in such a way that there is an explicit distinction on what part of the data is allowed to be released. The *when* dimension, on the other hand, consists on designing the trusted modules in such a way that certain privileges can only be exercised when some, well-defined, events occurs.

In our model, privileges can be *delegated.* Specifically, a process may delegate privileges to another process according to the following definition:

Definition 5 (Can-delegate relation). *A process with privilege P can delegate any privilege P', such that $P \implies P'$.*

In other words, it is possible to delegate a privilege P' that is at most as strong as the parent privilege P. In Section 5, we give a concrete example of using delegation to implement a privilege separation.

2.2 Ownership and categories

Our definition of DC label components as conjunctions of clauses, each imposing an information flow restriction, is similar to the DStar [31] label format which uses a set of *categories*, each of which is used to impose a flow restriction. Though the name category may be used interchangeably with clause, our categories differ from those of DStar (or even DLM) in that they are disjunctions of principals—hence the name, *disjunction category* labels.

The principals composing a category are said to *own* the category—every owner is trusted to uphold or bypass the restriction imposed by the category. For instance, the category [Bob ∨ Alice] is owned by both Alice and Bob. We can thus interpret the secrecy component $\{[\text{Bob} \vee \text{Alice}] \wedge \text{Preparer}\}$ to specify that data can be observed by the Preparer in collaboration with *either* Bob or Alice. Though implicit in our definition of a DC label, this joint ownership of a category allows for expressing quite complex policies. For example, to file joint taxes with Alice, Bob can simply labels the tax data $\langle\{[\text{Bob} \vee \text{Alice}]\}, \{\text{Bob}\}\rangle$, and now the WebTax results can be observed by both him and Alice. Expressing such policies in other systems, such as DLM or DStar, can only be done through external means (e.g., by creating a new principal AliceBob and encoding its relationship to Alice and Bob in a centralized principal hierarchy).

In the previous section we represent privileges P as a conjunction of principals for whom code is acting. Analogous to a principal owning a category, we say that a process (or computation) *owns* a principal if it acting or running on its behalf. (More generally, the code is said to own all the categories that compose P.)

3 Soundness

In this section, we show that the can-flow-to relation ($\sqsubseteq$) and the relation ($\sqsubseteq_P$) for can-flow-to given privileges P satisfy various properties. We first show that $\sqsubseteq$, given in Definition 2, is partial order.

Lemma 1 (DC labels form a partially ordered set). *The binary relation $\sqsubseteq$ over the set of all DC labels is a partial order.*

Proof. Reflexivity and transitivity follow directly from the Reflexivity and transitivity of ($\Longrightarrow$). By Definition 1, the components of a label, and thus the label, have a unique representation. Directly, the antisymmetry property holds.

Recall from Section 2 that for any two labels L_1 and L_2 there exists a join $L_1 \sqcup L_2$ and meet $L_1 \sqcap L_2$. The join must be the least upper bound of L_1 and L_2, with $L_1 \sqsubseteq L_1 \sqcup L_2$, and $L_2 \sqsubseteq L_1 \sqcup L_2$; conversely, the meet must be the greatest lower bound of L_1 and L_2, with $L_1 \sqcap L_2 \sqsubseteq L_1$ and $L_1 \sqcap L_2 \sqsubseteq L_2$. We prove these properties and show that DC labels form a lattice.

Proposition 1 (DC labels form a bounded lattice). *DC labels with the partial order relation $\sqsubseteq$, join $\sqcup$, and meet $\sqcap$ form a bounded lattice with minimum element $\bot = \langle \textbf{True}, \textbf{False} \rangle$ and maximum element $\top = \langle \textbf{False}, \textbf{True} \rangle$.*

Proof. The lattice property follows from Lemma 1, the definition of DC labels, and the definition of the join and meet as given in Definition 3.

It is worth noting that the DC label lattice is actually product lattice, i.e., a lattice where components are elements of a secrecy and (a dual) integrity lattice [29].

In Section 2.1 we detailed declassification and endorsement of data in terms of exercising privileges. Both actions constitute bypassing restrictions of $\sqsubseteq$ by using a more permissive relation $\sqsubseteq_P$. Here, we show that this privilege-exercising relation, as given in Definition 4, is a pre-order and that privilege delegation respects its restrictions.

Proposition 2 (The $\sqsubseteq_P$ relation is a pre-order). *The binary relation $\sqsubseteq_P$ over the set of all DC labels is a pre-order.*

Proof. Reflexivity and transitivity follow directly from the reflexivity and transitivity of ($\Longrightarrow$). Unlike $\sqsubseteq$, however, $\sqsubseteq_P$ is not necessarily antisymmetric (showing this, for a non-empty P, is trivial).

Informally, exercising privilege P may allow a principal to ignore the distinction between certain pairs of clauses, hence $\sqsubseteq_P$ is generally not a partial order. Moreover, the intuition that $\sqsubseteq_P$, for any non-empty P, is always more permissive than $\sqsubseteq$ follows as a special case of the following proposition.

Proposition 3 (Privileges substitution). *Given privileges P and P', if $P \Longrightarrow P'$ then P can always be substituted in for P'. Specifically, for all labels L_1 and L_2, if $P \Longrightarrow P'$ and $L_1 \sqsubseteq_{P'} L_2$ then $L_1 \sqsubseteq_P L_2$.*

Proof. First, we note that if $P \Longrightarrow P'$, then for any X, X', such that $X \wedge P' \Longrightarrow X'$, the proposition $X \wedge P \Longrightarrow X \wedge P' \Longrightarrow X'$ holds trivially. By Definition 4, $L_1 \sqsubseteq_{P'} L_2$ is equivalent to: $S_2 \wedge P' \Longrightarrow S_1$ and $I_1 \wedge P' \Longrightarrow I_2$. However, from $P \Longrightarrow P'$, we have $S_2 \wedge P \Longrightarrow S_2 \wedge P' \Longrightarrow S_1$, and $I_1 \wedge P \Longrightarrow I_1 \wedge P' \Longrightarrow I_2$. Correspondingly, we have $L_1 \sqsubseteq_P L_2$.

Informally, if a piece of code exercises privileges P' to read or endorse a piece of data, it can do so with P as well. In other words, $\sqsubseteq_P$ is at least at as permissive as $\sqsubseteq_{P'}$. Letting $P' = \textbf{True}$, it directly follows that for any non-empty P, i.e., for $P \neq \textbf{True}$, the relation $\sqsubseteq_P$ is more permissive than $\sqsubseteq$. Moreover, negating the statement of the proposition (if $L_1 \not\sqsubseteq_P L_2$ then $L_1 \not\sqsubseteq_{P'} L_2$) establishes that if exercising a privilege P does not allow for the flow of information from L_1 to L_2, then exercising a privilege delegated from P will also fail to allow the flow. This property is especially useful in guaranteeing soundness of privilege separation.

4 Model Extensions

The base DC label model, as described in Section 2, can be used to implement complex DIFC systems, despite its simplicity. Furthermore, the model can easily be further extended to support features of existing security (IFC and capability) systems, as we detail below.

4.1 Principal hierarchy

As previously mentioned, DLM [21] has a notion of a principal hierarchy defined by a reflexive and transitive relation, called *acts for*. Specifically, a principal p can act for another principal p', written $p \succeq p'$, if p is at least as powerful as p': p can read, write, declassify, and endorse all objects that p' can; the principal hierarchy tracks such relationships.

To incorporate this feature, we modify our model by encoding the principal hierarchy as a set of axioms Γ. Specifically, if $p \succeq p'$, then $(p \implies p') \in \Gamma$. Consequently, Γ is used as a hypothesis in every proposition. For example, without the principal hierarchy $\emptyset \vdash p_1 \implies [p_2 \vee p_3]$ does not hold, but if $p_1 \succeq p_2$ then $(p_1 \implies p_2), \Gamma \vdash p_1 \implies [p_2 \vee p_3]$ does hold. We, however, note that our notion of privileges and label component clauses (disjunction categories) can be used to capture such policies, that are expressible in DLM only through the use of the principal hierarchy. Compared to DLM, DC labels can be used to express very flexible policies (e.g., joint ownership) even when $\Gamma = \emptyset$.

4.2 Using DC labels in a distributed setting

For scalability, extending a system to a distributed setting is crucial. Addressing this issue, Zeldovich et al. [31], provide a distributed DIFC system, called DStar. DStar is a framework (and protocol) that extends OS-level DIFC systems to a distributed setting. Core to DStar is the notion of an *exporter* daemon, which, among other things, maps DStar network labels to OS local labels such as DC labels, and conversely. DC labels (and privileges) are a generalization of DStar labels (and privileges)—the core difference being the ability of DC labels to represent joint ownership of a category with disjunctions, a property expressible in DStar only with privileges. Hence, DC labels can directly be used when extending a system to a distributed setting. More interestingly, however, we can extend DStar, while remaining backwards compatible (since every DStar label can be expressed using a DC label), to use disjunction categories and thus, effectively, use DC labels as network labels—this extension is part of our future work.

4.3 Delegation and pseudo-principals

As detailed in Section 2.1, our decentralized privileges can be delegated and thus create a privilege hierarchy. Specifically, a process with a set of privileges may delegate a category it owns (in the form of a single-category privilege), which can then be further *granted* or delegated to another process.

In scenarios involving delegated privileges, we introduce the notion of a *pseudo-principal*. Pseudo-principals allows one to express providence on data, which is particularly useful in identifying the contributions of different computations to a task. A pseudo-principal is simply a principal (distinguished by the prefix #) that cannot be owned by any piece of code and can only be created when a privilege is delegated. Specifically, a process that owns principal p may delegate a single-category privilege $\{[p \vee \#c]\}$ to a piece of code c. The disjunction is used to indicate that the piece of code c is responsible for performing a task been delegated by the code owing p, which also does not trust c with the privilege p. Observe that the singleton $\{\#c\}$ cannot appear in any privilege, and as a result, if some data is given to p with the integrity restriction $[p \vee \#c]$, then the piece of code c must have been the originator. In a system with multiple components, using pseudo-principals, one can enforce a pipeline of operations, as shown by the implementation of a mail delivery agent in Section 6.

We note that pseudo-principals are treated as ordinary principals in label operations. Moreover, in our implementation, the distinction is minimal: principals are strings that cannot contain the character '`#`', while pseudo-principals are strings that always have the prefix '`#`'.

4.4 Privilege revocation

In dynamic systems, security policies change throughout the lifetime of the system. It is common for new users to be added and removed, as is for privileges to be granted and revoked [2]. Although our model can be extended to support revocation similar to that of public key infrastructures [11], we describe a selective revocation approach, common to capability-based systems [24].

To allow for the flexibility of selective revocation, it is necessary to keep track of a delegation chain with every category in a delegated privilege. For example, if processes A and C respectively delegate the single-principal privileges $\{a\}$ and $\{c\}$ to process B, B's privilege will be encoded as $\{(\{A \to B\}, a), (\{C \to B\}, c)\}$. Similarly, if B delegates $\{[a \vee c]\}$ to D, the latter's privilege set will be $\{(\{A \to B \to D, C \to B \to D\}, [a \vee c])\}$. Now, to selectively revoke a category, a process updates a system-wide revocation set Ψ with a pair consisting of the chain prefix and a privilege (it delegated) to be revoked. For example, A can revoke B's ownership of $\{a\}$ by adding $(\{A \to B\}, a)$ to Ψ. Consequently, when B or D perform a label comparison involving privileges, i.e., use $\sqsubseteq_P$, the revocation set Ψ is consulted: since $A \to B$ is a prefix in both cases, and $a \implies a$ and $a \implies [a \vee c]$, neither B nor D can exercise their delegated privileges. More generally, ownership of single-category privilege $\{c\}$ with chain x is revoked if there is a pair $(y, \psi) \in \Psi$ such that the chain y is a prefix of a chain in x and $\psi \implies c$. We finally note that, although this description of revocation relies on a centralized revocation set Ψ, selective revocation, in practice, can be implemented without a centralized set, using patterns such as Redell's "caretaker pattern" [24, 18] with wrapper, or membrane, objects transitively applying the revocation [19, 18].

5 Security Labeling Patterns

When building practical IFC systems, there are critical design decisions involving: (1) assigning labels to entities (data, channels, etc.), and (2) delegating privileges to executing code. In this section, we present *patterns* that can be used as a basis for these design decisions, illustrated using simplified examples of practical system applications.

5.1 Confinement and access control

A very common security policy is *confinement*: a program is allowed to compute on sensitive data but cannot export it [16, 26]. The tax-preparation example of Section 2 is a an examples of a system that enforces confinement.

In general, we may wish to confine a computation and guarantee that it does not release (by declassification) user A's sensitive data to the public network or any other channel. Using the network as an illustrative example, and assuming A's sensitive data is labeled L_A, confinement may be achieved by executing the computation with privileges P chosen such that $L_A \not\sqsubseteq_P L_{\text{pub}}$. A complication is that most existing IFC systems (though not all, see, e.g., [6, 14]) are susceptible to covert channel attacks that circumvent the restrictions based on labels and privileges. For example, a computation with no privileges might read sensitive data and leak information by, e.g., not terminating or affecting timing behavior. To address confinement in the presence of covert channels, we use the notion of *clearance* [5], previously introduced and formalized in [30, 30, 28] in the context of IFC.

Clearance imposes an upper bound on the sensitivity of the data that the computation can read. To prevent a computation from accessing (reading or writing) data labeled L_A, we set the computation's clearance to some L_C such that $L_A \not\sqsubseteq L_C$. With this restriction, the computation may read data labeled L_D only if $L_D \sqsubseteq L_C$. Observe that in a similar manner, clearance can be used to enforce other forms of discretionary access control.

5.2 Privilege separation

Using delegation, a computation may be compartmentalized into sub-computations, with the privileges of the computation subdivided so that each sub-computation runs with *least privilege*. Consider, for example, a privilege-separated mail delivery agent (MDA) that performs spam filtering.

As with many real systems, the example MDA of Fig. 2 is composed of different, and possibly untrustworthy, modules. In this example, the components are a network receiver, R, and a spam filter, S. Instead of combining the components into a monolithic MDA, the MDA author can segregate the untrustworthy components and execute then with the principle

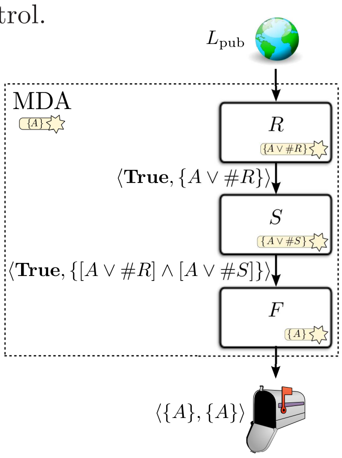

L_{pub}
MDA
$\{A\}$
R
$\{A \vee \#R\}$
$\langle \mathbf{True}, \{A \vee \#R\} \rangle$
S
$\{A \vee \#S\}$
$\langle \mathbf{True}, \{[A \vee \#R] \wedge [A \vee \#S]\} \rangle$
F
$\{A\}$
$\langle \{A\}, \{A\} \rangle$

Fig. 2: Simple MDA.

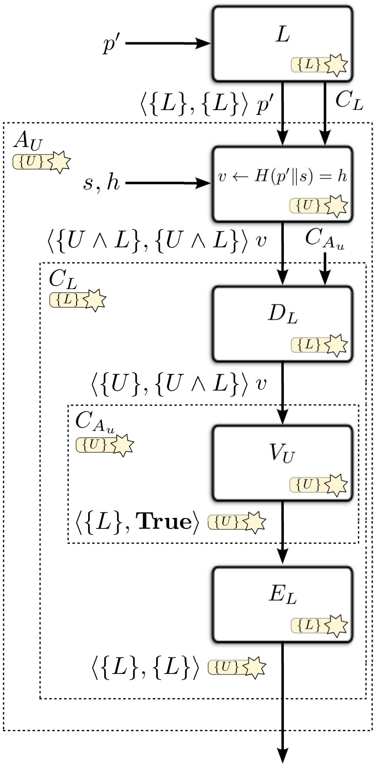
of least privilege. This avoids information leaks and corruption due to negligence or malfeasance on the component authors' part. Specifically, the receiver R is executed with the delegated privilege $\{[A \vee \#R]\}$, and the spam filter S is executed with the privilege $\{[A \vee \#S]\}$. As a consequence, R and S cannot read A's sensitive information and leak it to the network, corrupt A's mailbox, nor forge data on A's behalf.

Additionally, the MDA can enforce the policy that a mail message always passes through both receiver R and spam filter S. To this end, the MDA includes a small, trusted forwarder F, running with the privilege $\{A\}$, which endorses messages on behalf of A and writes them to the mailbox only after checking that they have been endorsed by both R and S. In a similar manner, this example can be further extended to verify that the provenance of a message is the network interface, or that the message took a specific path (e.g., R then S, but not S then R), among other.

5.3 User authentication

Another common requirement of security systems is user authentication. We consider password-based login as an example, where a successful authentication corresponds to granting the authenticated user the set of privileges associated with their credentials. Furthermore, we consider authentication in the context of (typed) language-level DIFC systems; an influential OS-level approach has been considered in [30]. Shown in Fig. 3 is an example system which consists of a login client L, and an authentication service A_U.

To authenticate user U, the login client *invokes* the user authentication service A_U, which runs with the $\{U\}$ privilege. Conceptually, when invoked with U's correct credentials, A_U grants (by delegating) the caller the $\{U\}$ privilege. However, in actuality, the login client and authentication service are in mutual distrust: L does not trust A_U with U's password, for A_U might be malicious and simply wish to learn the password, while A_U does not trust L to grant it the $\{U\}$ privilege without first verifying credentials. Consequently, the authentication requires several steps.



p'
L
$\{L\}$
$\langle\{L\},\{L\}\rangle\ p'$
C_L
A_U
$\{U\}$
s, h
$v \leftarrow H(p'\|s) = h$
$\{U\}$
$\langle\{U \wedge L\},\{U \wedge L\}\rangle\ v$
C_{A_u}
C_L
$\{L\}$
D_L
$\{L\}$
$\langle\{U\},\{U \wedge L\}\rangle\ v$
C_{A_u}
$\{U\}$
V_U
$\{U\}$
$\langle\{L\},\mathbf{True}\rangle$
$\{U\}$
E_L
$\{L\}$
$\langle\{L\},\{L\}\rangle$
$\{U\}$

Fig. 3: User authentication.

We note that due to the mutual distrust, the user's stored salt s and password hash $h = H(p\|s)$ is labeled with both, the user and login client's, principals, i.e., h and s have label $\langle\{U \wedge L\},\{U \wedge L\}\rangle$. Solely, labeling them $\langle\{U\},\{U\}\rangle$ would allow A_U to carry out an off-line attack to recover p. The authentication procedure is as follows.

1. The user's input password p' to the login client is labeled $\langle \{L\}, \{L\} \rangle$, and along with a closure C_L is passed to the authentication service A_U. As further detailed below, closures are used in this example as a manner to exercise privileges under particular conditions and operations.
2. A_U reads U's stored salt s and password hash h. It then computes the hash $h' = H(p' \| s)$ and compares h' with h. The label of this result is simply the join of h and h': $\langle \{U \wedge L\}, \{L\} \rangle$. Since A_U performed the computation, it endorses the result by adding U to its integrity component; for clarity, we name this result v, as show in Fig 3.

- *Remark:* At this point, neither L nor A_U are able to read and fully declassify the secret password-check result v. Moreover, without eliminating the mutual distrust, neither L nor S can declassify v directly. Consider, for example, if A_U is malicious and had, instead, performed a comparison of $H(p' \| s)$ and $H(p'' \| s)$, for some guessed password p''. If L were to declassify the result, A_U would learn that $p = p''$, assuming the user typed in the correct password, i.e., $p = p'$. Hence, we rely on purely functional (and statically-typed) closures to carry out the declassification indirectly.

3. When invoking A_U, L passed a declassification closure C_L, which has the $\{L\}$ privilege locally bound. Now, A_U invokes C_L with v and its own declassification closure C_{A_U}.
4. C_L declassifies v (D_L in Fig. 3) to $\langle \{U\}, \{U \wedge L\} \rangle$, and then invokes C_{A_U} with the new, partially-declassified result.
5. The C_{A_U} closure has the $\{U\}$ privilege bound and upon being invoked, simply verifies the result and its integrity (V_U in Fig. 3). If the password is correct v is true and then C_{A_U} returns the privilege $\{U\}$ labeled with $\langle \{L\}, \textbf{True} \rangle$; otherwise it returns the empty privilege set. It is important that the integrity of v be verified, for a malicious L could provide a closure that forges password-check results, an attempt to wrongfully gain privileges.
6. The privilege returned from invoking C_{A_U} is endorsed by C_L (E_L in Fig. 3), only if its secrecy component is L. This asserts that upon returning the privilege from C_L, A_U cannot check if the privilege is empty or not, and thus infer the comparison result.
7. It only remains for A_U to forward the labeled privilege back to L.

We finally note that the authentication service is expected to keep state that tracks the number of attempts made by a login client, as each result leaks a bit of information; to limit the number of unsuccessful attempts requires the use of a (minimal) code that is trusted by both L and A_U, as shown in [30].

6 Implementing DC labels

We present two Haskell implementations of DC labels[3]. The first, `dclabel`, is a library that provides a simple embedded domain specific language for constructing and working with dynamic labels and privileges. Principals in the `dclabel`

[3] Available at `http://www.scs.stanford.edu/~deian/dclabels`

library are represented by strings, while label components are lists of clauses (categories), which, in turn, are lists of principals. We use lists as sets for simplicity and because Haskell supports list comprehension; this allowed for a very simple translation from the formal definitions of this paper to (under 180 lines of) Haskell code. We additionally implemented the instances necessary to use DC labels with the label-polymorphic dynamic DIFC library, LIO [28]. Given the simplicity of the implementation, we believe that porting it to other libraries, such as [17, 13], can be accomplished with minimal effort. Finally, we note that our implementation was thoroughly tested using the QuickCheck[4] library, however formal verification of the implementation using Zeno [27], a Haskell automated proof system, was unsuccessful. This is primarily due to Zeno's infancy and lack of support for analyzing Haskell list comprehension. A future direction includes implementing DC labels in Isabelle or Coq from which a provably-correct Haskell implementation can be extracted.

Although we have primarily focused on dynamic IFC, in cases where covert channels, runtime overhead, or failures are not tolerable, DC labels can also be used to enforce IFC statically. To this end, we implement `dclabel-static`, a prototype IFC system that demonstrates the feasibility of statically enforcing DIFC using secrecy-only DC labels, without modifying the Haskell language or the GHC compiler. Since DC labels are expressed using propositional logic, a programming language that has support for sum, product, and function types can be used, *without modification*, to enforce information flow control according to the Curry-Howard correspondence [12, 9]. According to the correspondence, disjunction, conjunction and implication respectively correspond to sum, product, and function types. Hence, for a secrecy-only DC label, to prove $L_1 \sqsubseteq L_2$, i.e., $L_2 \implies L_1$, we need only construct a function that has type $L_2 \to L_1$: successfully type-checking a program directly corresponds to verifying that the code does not violate IFC.

The library exports various type classes and combinators that facilitates the enforcement of static IFC. For example, we provide type constructors to create labels from principals—a principal in this system is a type for which an instance of the `Principal` type class is defined. To label values, we associate labels with types. Specifically, a labeled type is a wrapper for a product type, whose first component is a label, and whose second component, the value, cannot be projected without declassification. The library further provides a function, `relabel`, which, given a labeled value (e.g., $(L_1, 3)$), a new label L_2, and a proof of $L_1 \sqsubseteq L_2$ (a lambda term of type $L_2 \to L_1$), returns the relabeled value (e.g., $(L_2, 3)$). Since providing such proofs is often tedious, we supply a tool called `dcAutoProve`, that automatically inserts proofs of can-flow-to relations for expressions named `auto`, with an explicit type signature. Our automated theorem prover is based a variant of Gentzen's LJ sequent calculus [7].

[4] `http://hackage.haskell.org/package/QuickCheck`

7 Related work

DC labels closely resemble DLM labels [21] and their use in Jif [22]. Like DC labels, DLM labels express both secrecy and integrity policies. Core to a DLM label are components that specify an owner (who can declassify the data) and a set of readers (or writers). Compared to our disjunction categories, DLM does not allow for joint ownership of a component—they rely on a centralized principal hierarchy to express partial ownership. However, policies (natural to DLM) which allow for multiple readers, but a single owner, in our model, require a labeling pattern that relies on the notion of clearance, as discussed in Section 5 and used in existing DIFC systems [30, 31, 28]. Additionally, unlike to DLM labels as formalized in [20], DC labels form a bounded lattice with a join and meet that respectively correspond to the least upper bound and greatest lower bound; the meet for DLM labels is not always the greatest lower bound.

The language Paralocks [3] uses Horn clauses to encode fine-grained IFC policies following the notion of locks: certain flows are allowed when corresponding locks are open. Constraining our model to the case where a privilege set is solely a conjunction of principals, Paralocks be easily used to encode our model. However, it remain an open problem to determine if disjunctive privileges can be expressed in their notion of *state*.

The Asbestos [8] and HiStar [30] operating systems enforce DIFC using Asbestos labels. Asbestos labels use the notion of categories to specify information flow restrictions in a similar manner to our clauses/categories. Unlike DC labels, however, Asbestos labels do not rely on the notion of principals. We can map a subset of DC labels to Asbestos labels by mapping secrecy and integrity categories to Asbestos levels **3** and **0**, respectively. Similarly ownership of a category maps to level ⋆. This mapping is limited to categories with no disjunction, which are equivalent to DStar labels [31], as discussed in Section 4. Mapping disjunction categories can be accomplished by using the system's notion of privileges. Conversely, both Asbestos and DStar labels are subsumed by our model. Moreover, compared to these systems we give precise semantics, prove soundness of the label format, and show its use in enforcing DIFC statically.

Capability-based systems such as KeyKOS [1], and E [19] are often used to restrict access to data. Among other purposes, capabilities can be used to enforce discretionary access control (DAC), and though they can enforce MAC using patterns such as membranes, the capability model is complimentary. For instance, our notion of privilege is a capability, while a delegated privilege loosely corresponds to an attenuated capability. This notion of privileges as capabilities is like that of Flume [15]. However, whereas they consider two types of privilege (essentially one for secrecy and another for integrity), our notion of privilege directly corresponds to ownership and conferring the right to exercise it in any way. Moreover, delegated privileges and the notion of disjunction provides an equal abstraction.

8 Conclusion

Decentralized information flow control can be used to build applications that enforce end-to-end security policies using untrusted code. DIFC systems rely on labels to track and enforce information flow. We present disjunction category labels, a new label format useful in enforcing information flow control in systems with mutually distrusting parties. In this paper, we give precise semantics for DC labels and prove various security properties they satisfy. Furthermore, we introduce and prove soundness of decentralized privileges that are used in declassifying and endorsing data. Compared to Myers and Liskov's DLM, our model is simpler and does not rely on a centralized principal hierarchy, our privilege hierarchy is distributed. We highlight the expressiveness of DC labels by providing several common design and labeling patterns. Specifically, we show how to employ DC labels to express confinement, access control, privilege separation, and authentication. Finally, further illustrating flexibility of the model, we describe two Haskell implementations: a library used to perform dynamic label checks, compatible with existing DIFC systems, and a prototype library that enforces information flow statically by leveraging Haskell's module and type system.

Acknowledgments This work was supported by DARPA CRASH and PROCEED, Google, the Swedish research agency VR, the NSF, and the AFOSR. D. Stefan is supported by the DoD through the NDEG Fellowship Program.

References

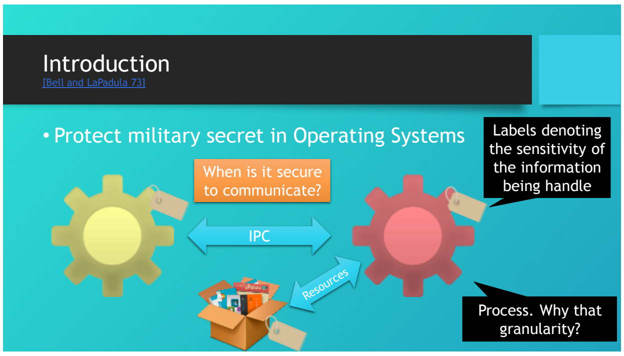
1. A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992.
2. D. Boneh, X. Ding, G. Tsudik, and C. Wong. A method for fast revocation of public key certificates and security capabilities. In *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10*, pages 22–22. USENIX Association, 2001.
3. N. Broberg and D. Sands. Paralocks: role-based information flow control and beyond. In *SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 431–444, 2010.
4. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
5. Department of Defense. *Trusted Computer System Evaluation Criteria (Orange Book)*, DoD 5200.28-STD edition, December 1985.
6. D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *2010 IEEE Symposium on Security and Privacy*, pages 109–124. IEEE, 2010.
7. R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, pages 795–807, 1992.
8. 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. of the 20th ACM Symposium on Operating Systems Principles*, pages 17–30, Brighton, UK, October 2005. ACM.
9. J. Gallier. Constructive logics part i: A tutorial on proof systems and typed λ-calculi. *Theoretical computer science*, 110(2):249–339, 1993.
10. J. Goguen and J. Meseguer. Security policies and security models. In I. C. S. Press, editor, *Proc of IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

11. C. Gunter and T. Jim. Generalized certificate revocation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 316–329. ACM, 2000.
12. W. Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pages 479–490, 1980.
13. M. Jaskelioff and A. Russo. Secure multi-execution in Haskell. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer Verlag, June 2011.
14. V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing-and termination-sensitive secure information flow: Exploring a new approach. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 413–428. IEEE, 2011.
15. 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. of the 21st Symp. on Operating Systems Principles*, October 2007.
16. B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.
17. P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.
18. M. Miller and J. Shapiro. Paradigm regained: Abstraction mechanisms for access control. *Advances in Computing Science–ASIAN 2003*, pages 224–242, 2003.
19. M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.
20. A. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *IEEE Security and Privacy, 1998.*, pages 186–197. IEEE, 1998.
21. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, 1997.
22. A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. on Computer Systems*, 9(4):410–442, October 2000.
23. C. Papadimitriou. *Complexity Theory*. Addison Wesley, 1993.
24. D. Redell and R. Fabry. Selective revocation of capabilities. In *Proceedings of the International Workshop on Protection in Operating Systems*, pages 192–209, 1974.
25. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.
26. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9):1278–1308, September 1975.
27. W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: A tool for the automatic verification of algebraic properties of functional programs. Technical report, Imperial College London, Feb. 2011.
28. D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*, pages 95–106. ACM SIGPLAN, September 2011.
29. S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.
30. N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.
31. N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proc. of the 6th Symp. on Networked Systems Design and Implementation*, pages 293–308, San Francisco, CA, April 2008.

Web Browser Security

Mandatory Access Control

Alejandro Russo
russo@chalmers.se
ECI 2015, UBA, Buenos Aires, Argentina



Introduction

[Bell and LaPadula 73]

- Protect military secret in Operating Systems

Labels denoting the sensitivity of the information being handle
When is it secure to communicate?
IPC
Resources
Process. Why that granularity?

Modern OS manifestation of MAC

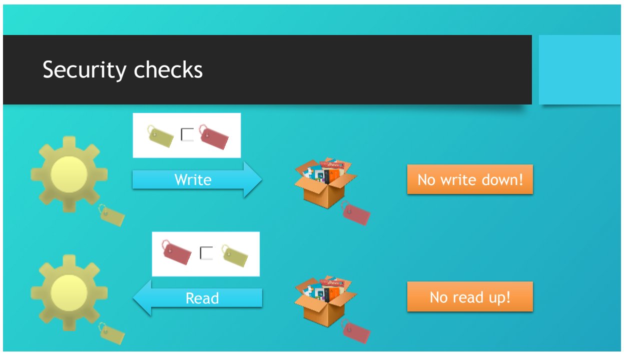
[Asbestos 2005]

[Flume 2007]

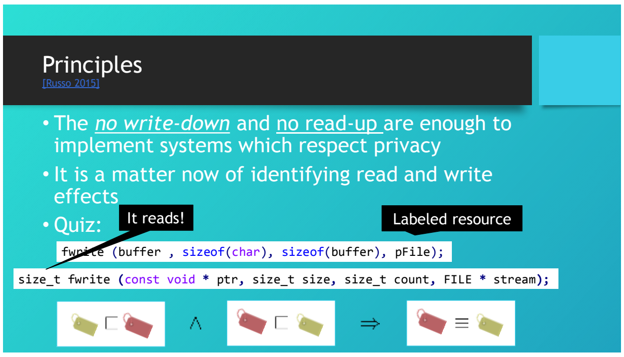
[HiStar 2006]

[seL4 2013]

CAUTION
Asbestos Containing Material (ACM)
Cancer and lung disease hazard
Do not disturb without proper training and equipment
NO LOGO
seL4



Security checks

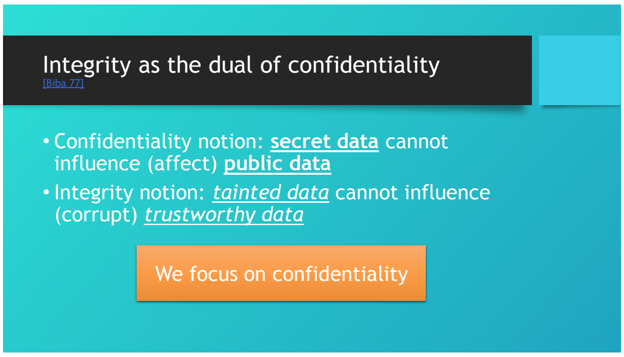


Principles

[Russo 2015]

- The *no write-down* and no read-up are enough to implement systems which respect privacy
- It is a matter now of identifying read and write effects
- Quiz:

```
fwrite (buffer , sizeof(char), sizeof(buffer), pFile);
size_t fwrite (const void * ptr, size_t size, size_t count, FILE * stream);
```

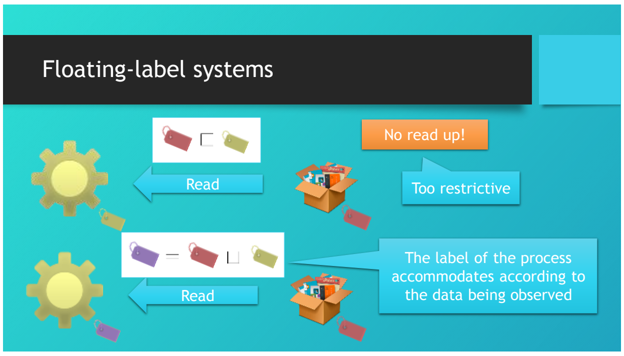


Integrity as the dual of confidentiality

[Biba 77]

- Confidentiality notion: **secret data** cannot influence (affect) **public data**
- Integrity notion: *tainted data* cannot influence (corrupt) *trustworthy data*

We focus on confidentiality



Floating-label systems

⊏

Read

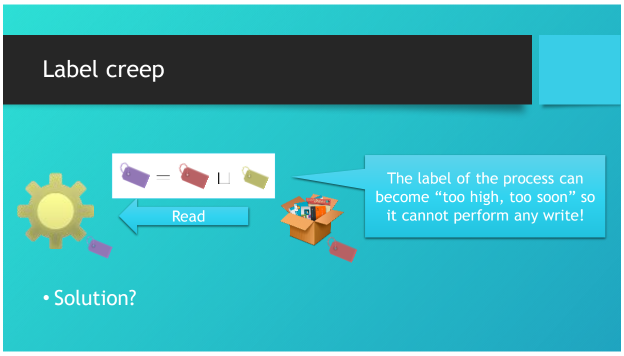
No read up!

Too restrictive

— ⊔

Read

The label of the process accommodates according to the data being observed



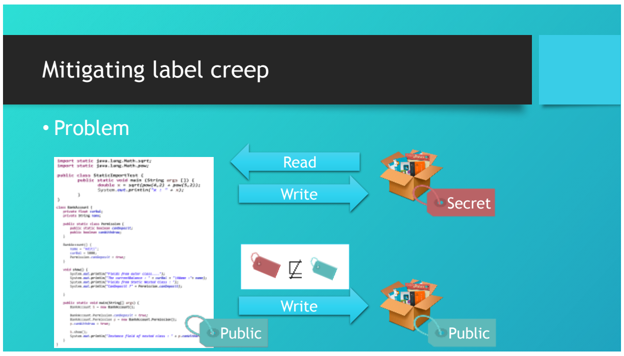
Label creep

— ⊔

Read

The label of the process can become “too high, too soon” so it cannot perform any write!

- Solution?



Mitigating label creep

- Problem

Read

Write

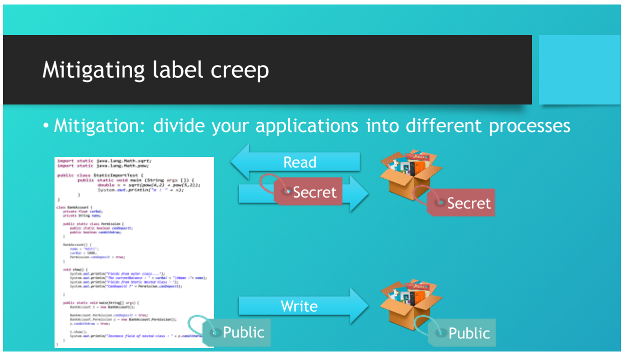
Secret

⋢

Write

Public

Public



Mitigating label creep

- Mitigation: divide your applications into different processes

Read

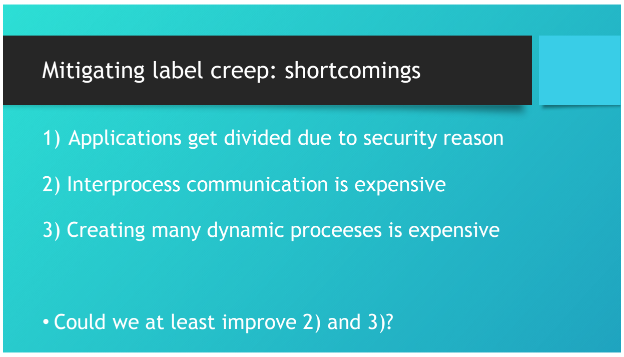
Secret

Secret

Write

Public

Public



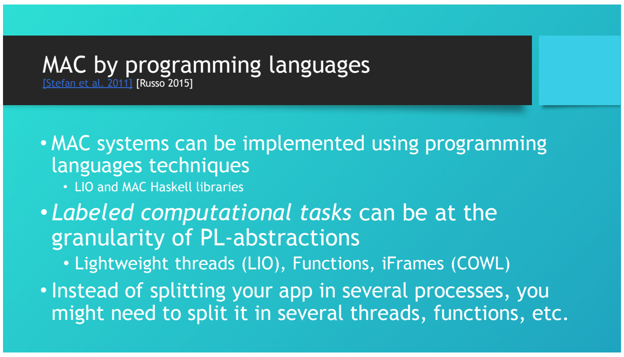
Mitigating label creep: shortcomings

1) Applications get divided due to security reason

2) Interprocess communication is expensive

3) Creating many dynamic proceeses is expensive

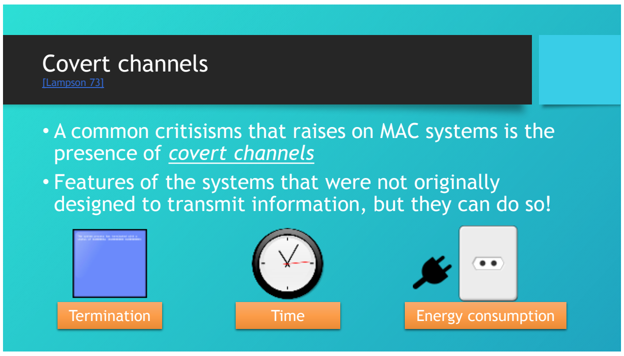
- Could we at least improve 2) and 3)?



MAC by programming languages

[Stefan et al. 2011] [Russo 2015]

- MAC systems can be implemented using programming languages techniques
  - LIO and MAC Haskell libraries
- *Labeled computational tasks* can be at the granularity of PL-abstractions
  - Lightweight threads (LIO), Functions, iFrames (COWL)
- Instead of splitting your app in several processes, you might need to split it in several threads, functions, etc.



Covert channels

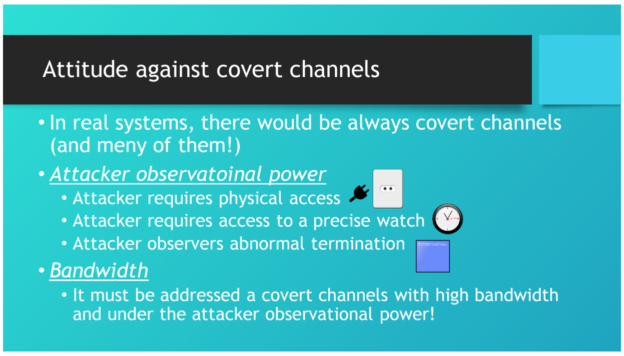
[Lampson 73]

- A common critisisms that raises on MAC systems is the presence of *covert channels*
- Features of the systems that were not originally designed to transmit information, but they can do so!

Termination

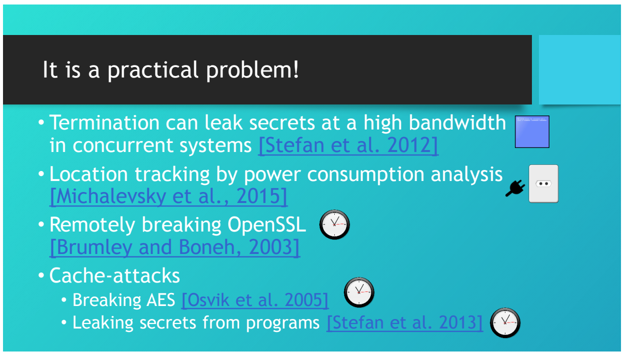
Time

Energy consumption



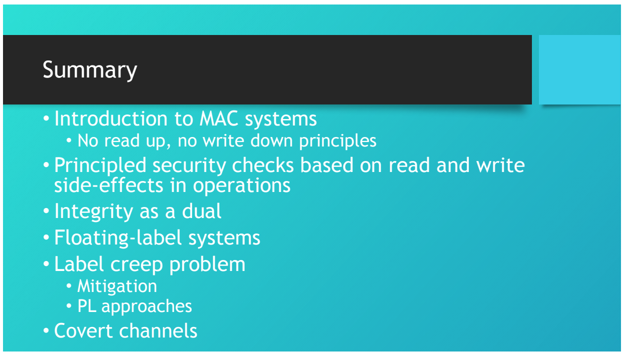
Attitude against covert channels

- In real systems, there would be always covert channels (and meny of them!)
- *Attacker observatoinal power*
  - Attacker requires physical access
  - Attacker requires access to a precise watch
  - Attacker observers abnormal termination
- *Bandwidth*
  - It must be addressed a covert channels with high bandwidth and under the attacker observational power!



It is a practical problem!

- Termination can leak secrets at a high bandwidth in concurrent systems [Stefan et al. 2012]
- Location tracking by power consumption analysis [Michalevsky et al., 2015]
- Remotely breaking OpenSSL [Brumley and Boneh, 2003]
- Cache-attacks
  - Breaking AES [Osvik et al. 2005]
  - Leaking secrets from programs [Stefan et al. 2013]



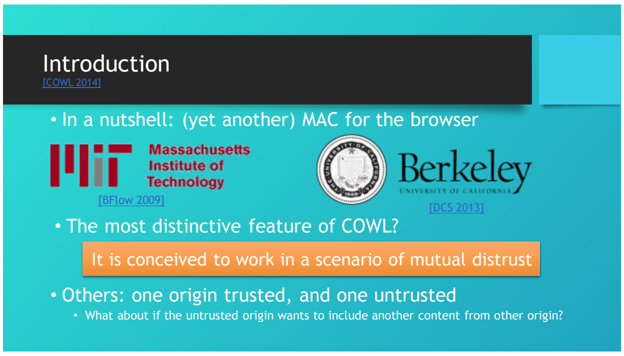
Summary

- Introduction to MAC systems
  - No read up, no write down principles
- Principled security checks based on read and write side-effects in operations
- Integrity as a dual
- Floating-label systems
- Label creep problem
  - Mitigation
  - PL approaches
- Covert channels

COWL

COWL (Firefox)

Alejandro Russo
russo@chalmers.se
ECI 2015, UBA, Buenos Aires, Argentina



Introduction

[COWL 2014]

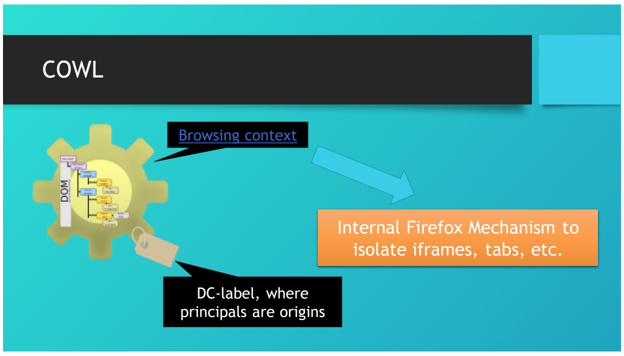
- In a nutshell: (yet another) MAC for the browser

Massachusetts Institute of Technology
[BFlow 2009]
Berkeley UNIVERSITY OF CALIFORNIA
[DCS 2013]

- The most distinctive feature of COWL?

It is conceived to work in a scenario of mutual distrust

- Others: one origin trusted, and one untrusted
  - What about if the untrusted origin wants to include another content from other origin?

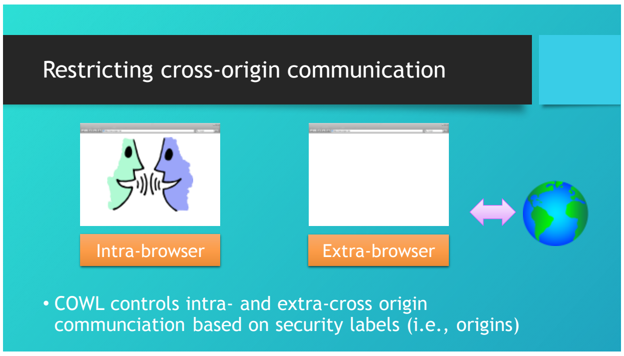


COWL

Browsing context

Internal Firefox Mechanism to isolate iframes, tabs, etc.

DC-label, where principals are origins

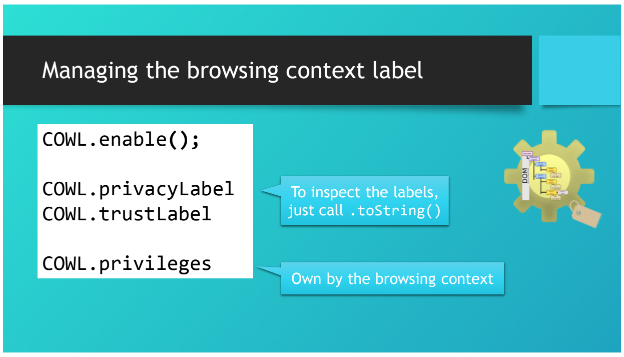


Restricting cross-origin communication

Intra-browser

Extra-browser

- COWL controls intra- and extra-cross origin communciation based on security labels (i.e., origins)

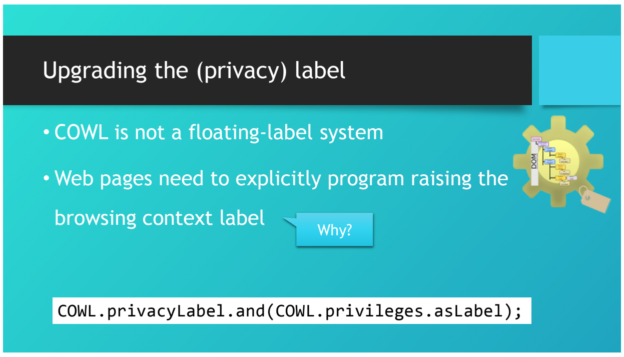


Managing the browsing context label

```
COWL.enable();

COWL.privacyLabel
COWL.trustLabel

COWL.privileges
```

To inspect the labels, just call .toString()

Own by the browsing context

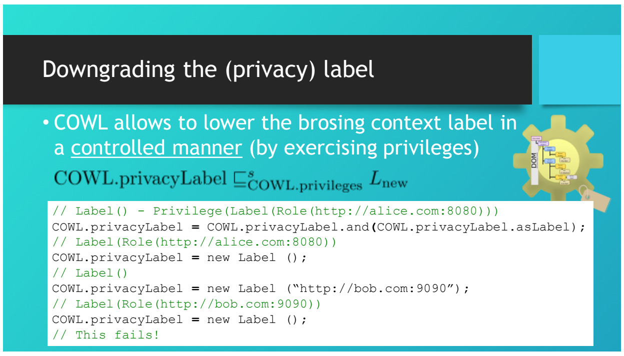


Upgrading the (privacy) label

- COWL is not a floating-label system
- Web pages need to explicitly program raising the browsing context label

Why?

```
COWL.privacyLabel.and(COWL.privileges.asLabel);
```

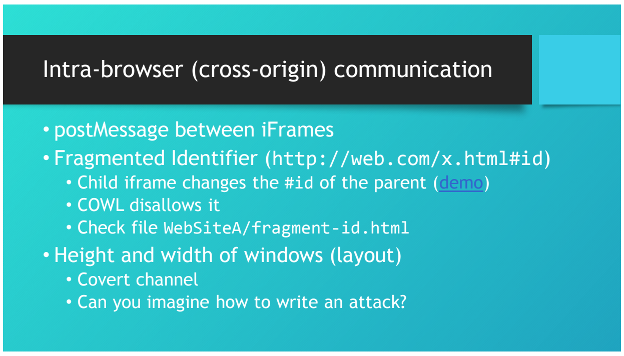


Downgrading the (privacy) label

- COWL allows to lower the brosing context label in a controlled manner (by exercising privileges)

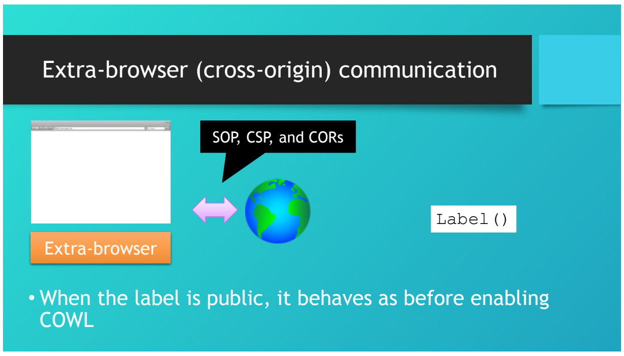
$\text{COWL.privacyLabel} \sqsubseteq^{s}_{\text{COWL.privileges}} L_{\text{new}}$

```
// Label() - Privilege(Label(Role(http://alice.com:8080)))
COWL.privacyLabel = COWL.privacyLabel.and(COWL.privacyLabel.asLabel);
// Label(Role(http://alice.com:8080))
COWL.privacyLabel = new Label ();
// Label()
COWL.privacyLabel = new Label ("http://bob.com:9090");
// Label(Role(http://bob.com:9090))
COWL.privacyLabel = new Label ();
// This fails!
```



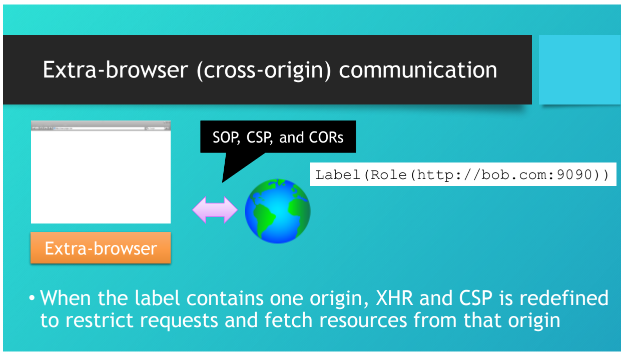
Intra-browser (cross-origin) communication

- postMessage between iFrames
- Fragmented Identifier (`http://web.com/x.html#id`)
  - Child iframe changes the `#id` of the parent (demo)
  - COWL disallows it
  - Check file `WebSiteA/fragment-id.html`
- Height and width of windows (layout)
  - Covert channel
  - Can you imagine how to write an attack?



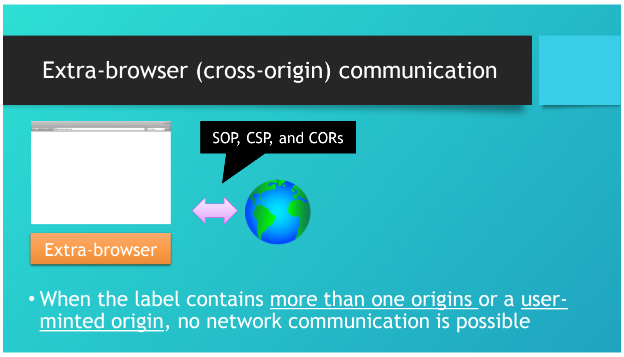
Extra-browser (cross-origin) communication

- When the label is public, it behaves as before enabling COWL



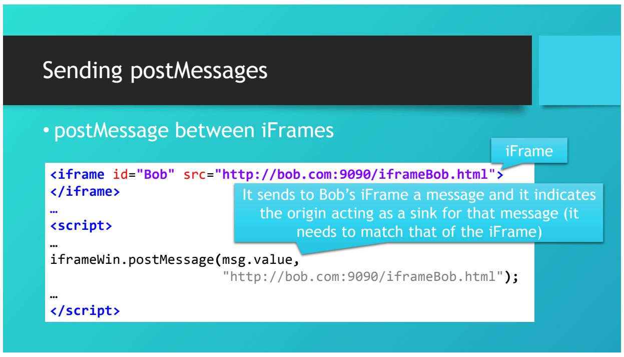
Extra-browser (cross-origin) communication

- When the label contains one origin, XHR and CSP is redefined to restrict requests and fetch resources from that origin



Extra-browser (cross-origin) communication

- When the label contains more than one origins or a user-minted origin, no network communication is possible



Sending postMessages

- postMessage between iFrames

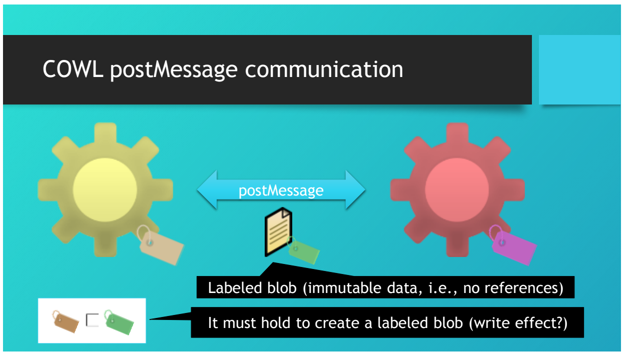
```
<iframe id="Bob" src="http://bob.com:9090/iframeBob.html">
</iframe>
…
<script>
…
iframeWin.postMessage(msg.value,
                      "http://bob.com:9090/iframeBob.html");
…
</script>
```

iFrame

It sends to Bob's iFrame a message and it indicates the origin acting as a sink for that message (it needs to match that of the iFrame)

Demo

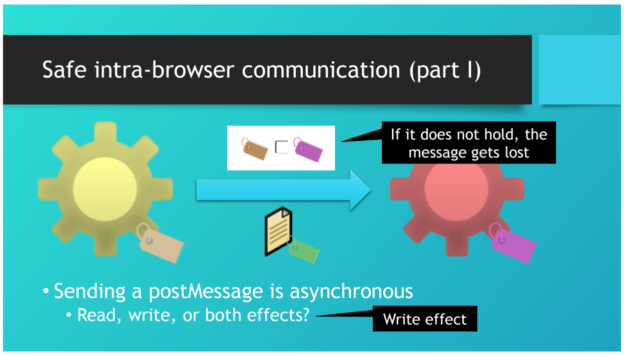
Files: `WebsiteA/iframeAlice.html`



COWL postMessage communication

Labeled blob (immutable data, i.e., no references)

It must hold to create a labeled blob (write effect?)

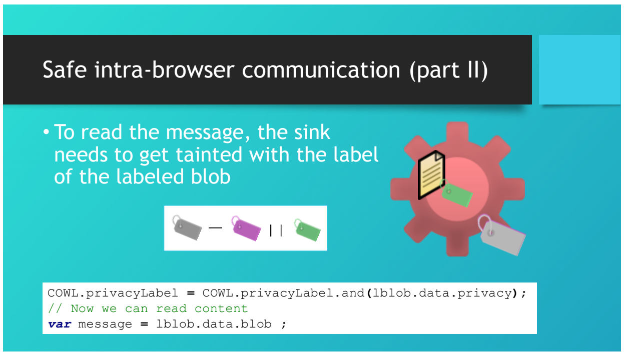


Safe intra-browser communication (part I)

If it does not hold, the message gets lost

- Sending a postMessage is asynchronous
  - Read, write, or both effects?

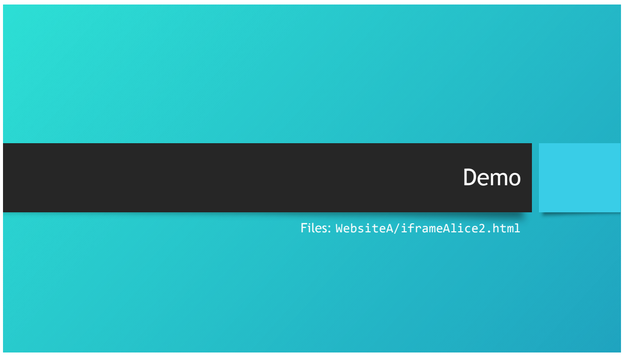
Write effect



Safe intra-browser communication (part II)

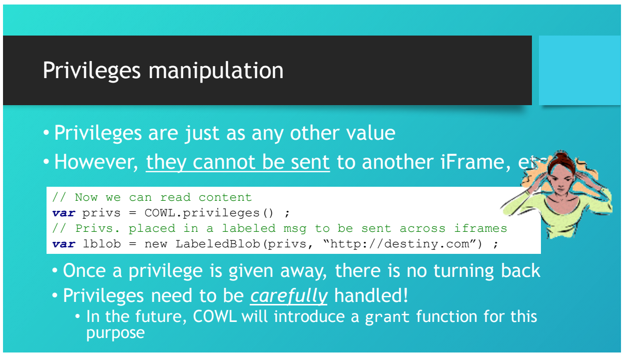
- To read the message, the sink needs to get tainted with the label of the labeled blob

```
COWL.privacyLabel = COWL.privacyLabel.and(lblob.data.privacy);
// Now we can read content
var message = lblob.data.blob ;
```



Demo

Files: `WebsiteA/iframeAlice2.html`

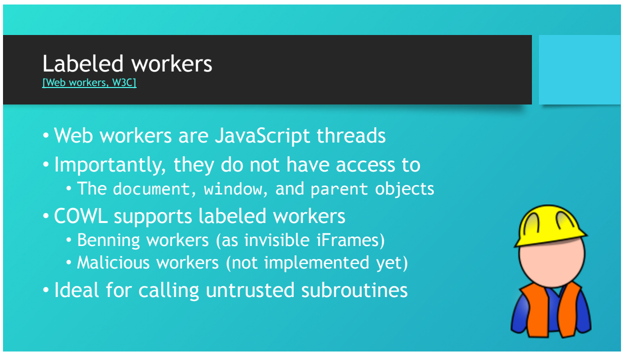


Privileges manipulation

- Privileges are just as any other value
- However, <u>they cannot be sent</u> to another iFrame, etc

```
// Now we can read content
var privs = COWL.privileges() ;
// Privs. placed in a labeled msg to be sent across iframes
var lblob = new LabeledBlob(privs, “http://destiny.com”) ;
```

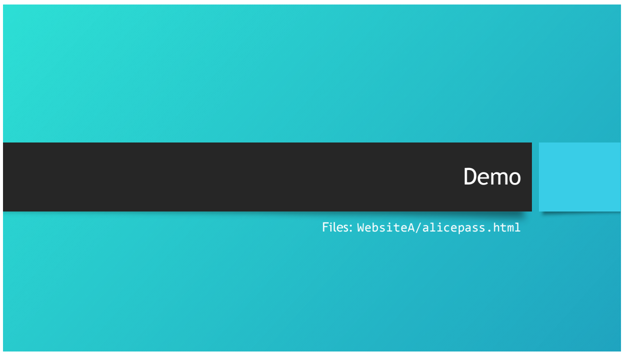
- Once a privilege is given away, there is no turning back
- Privileges need to be *<u>carefully</u>* handled!
  - In the future, COWL will introduce a `grant` function for this purpose



Labeled workers

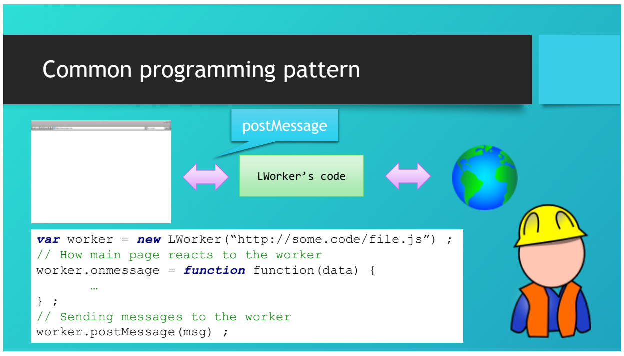
[Web workers, W3C]

- Web workers are JavaScript threads
- Importantly, they do not have access to
  - The `document`, `window`, and `parent` objects
- COWL supports labeled workers
  - Benning workers (as invisible iFrames)
  - Malicious workers (not implemented yet)
- Ideal for calling untrusted subroutines



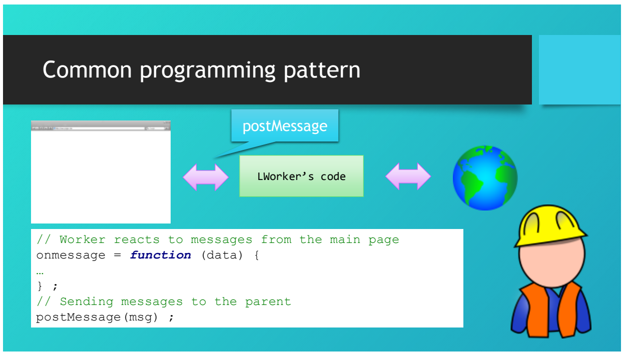
Demo

Files: `WebsiteA/alicepass.html`



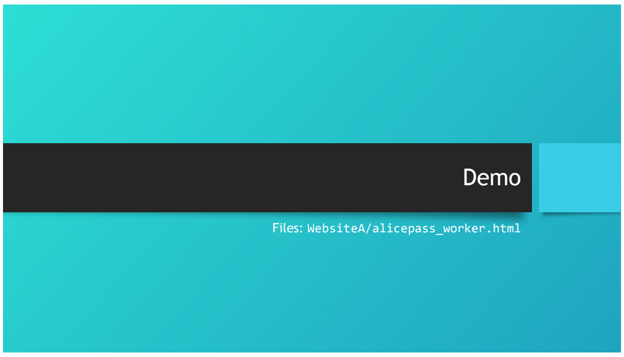
Common programming pattern

```
var worker = new LWorker("http://some.code/file.js") ;
// How main page reacts to the worker
worker.onmessage = function function(data) {
      …
} ;
// Sending messages to the worker
worker.postMessage(msg) ;
```



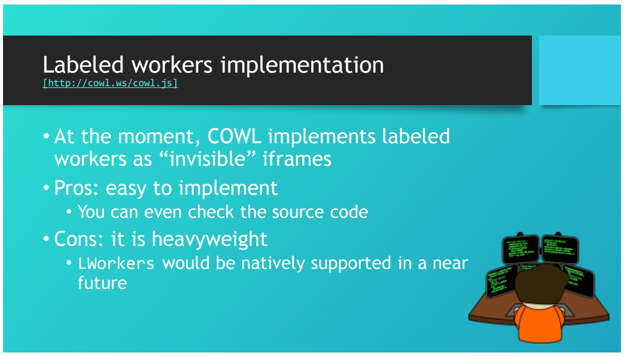
Common programming pattern

```
// Worker reacts to messages from the main page
onmessage = function (data) {
…
} ;
// Sending messages to the parent
postMessage(msg) ;
```



Demo

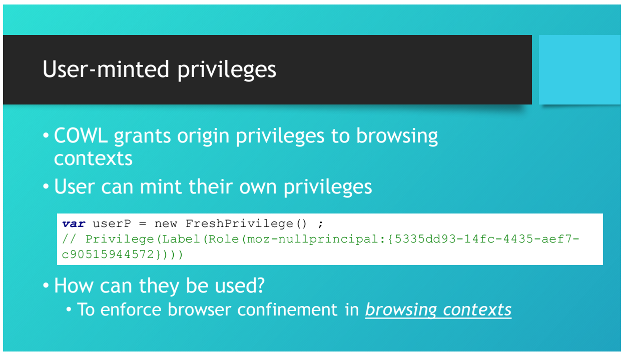
Files: `WebsiteA/alicepass_worker.html`



Labeled workers implementation

[http://cowl.ws/cowl.js]

- At the moment, COWL implements labeled workers as "invisible" iframes
- Pros: easy to implement
  - You can even check the source code
- Cons: it is heavyweight
  - `LWorkers` would be natively supported in a near future

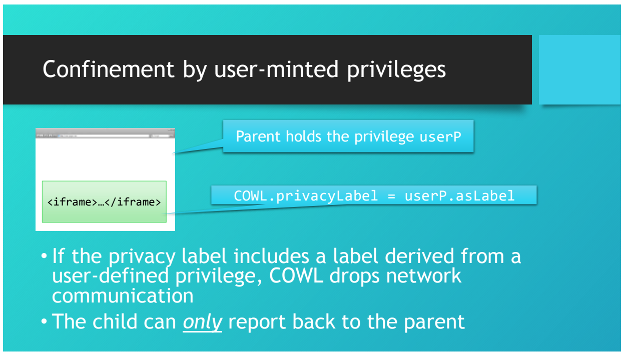


User-minted privileges

- COWL grants origin privileges to browsing contexts
- User can mint their own privileges

```
var userP = new FreshPrivilege() ;
// Privilege(Label(Role(moz-nullprincipal:{5335dd93-14fc-4435-aef7-c90515944572})))
```

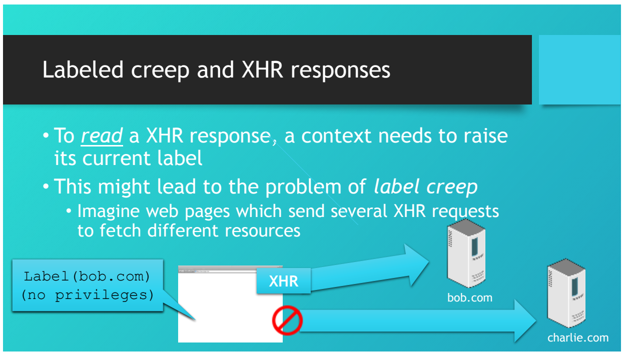
- How can they be used?
  - To enforce browser confinement in ***browsing contexts***



Confinement by user-minted privileges

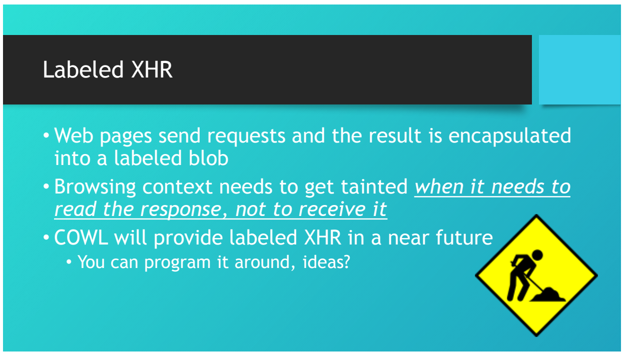
Parent holds the privilege userP
<iframe>…</iframe>
COWL.privacyLabel = userP.asLabel

- If the privacy label includes a label derived from a user-defined privilege, COWL drops network communication
- The child can ***only*** report back to the parent



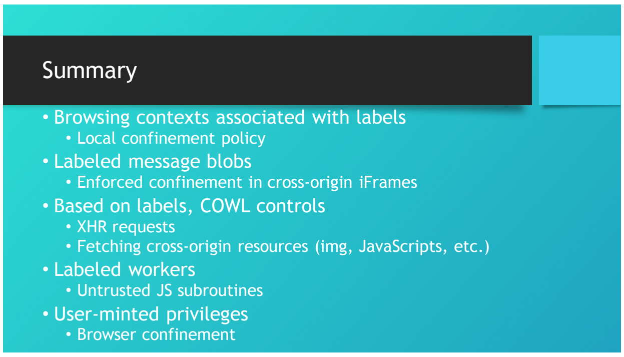
Labeled creep and XHR responses

- To ***read*** a XHR response, a context needs to raise its current label
- This might lead to the problem of *label creep*
  - Imagine web pages which send several XHR requests to fetch different resources



Labeled XHR

- Web pages send requests and the result is encapsulated into a labeled blob
- Browsing context needs to get tainted ***when it needs to read the response, not to receive it***
- COWL will provide labeled XHR in a near future
  - You can program it around, ideas?



Summary

- Browsing contexts associated with labels
  - Local confinement policy
- Labeled message blobs
  - Enforced confinement in cross-origin iFrames
- Based on labels, COWL controls
  - XHR requests
  - Fetching cross-origin resources (img, JavaScripts, etc.)
- Labeled workers
  - Untrusted JS subroutines
- User-minted privileges
  - Browser confinement

Protecting Users by Confining JavaScript with COWL

Deian Stefan[*] Stanford
Edward Z. Yang Stanford
Petr Marchenko Google
Alejandro Russo[†] Chalmers
Dave Herman Mozilla
Brad Karp UCL
David Mazières Stanford

Abstract

Modern web applications are conglomerations of JavaScript written by multiple authors: application developers routinely incorporate code from third-party libraries, and *mashup* applications synthesize data and code hosted at different sites. In current browsers, a web application's developer and user must trust third-party code in libraries not to leak the user's sensitive information from within applications. Even worse, in the status quo, the only way to implement some mashups is for the user to give her login credentials for one site to the operator of another site. Fundamentally, today's browser security model trades privacy for flexibility because it lacks a sufficient mechanism for *confining untrusted code.* We present COWL, a robust JavaScript confinement system for modern web browsers. COWL introduces label-based mandatory access control to browsing contexts in a way that is fully backward-compatible with legacy web content. We use a series of case-study applications to motivate COWL's design and demonstrate how COWL allows both the inclusion of untrusted scripts in applications and the building of mashups that combine sensitive information from multiple mutually distrusting origins, all while protecting users' privacy. Measurements of two COWL implementations, one in Firefox and one in Chromium, demonstrate a virtually imperceptible increase in page-load latency.

1 Introduction

Web applications have proliferated because it is so easy for developers to reuse components of existing ones. Such reuse is ubiquitous. jQuery, a widely used JavaScript library, is included in and used by over 77% of the Quantcast top-10,000 web sites, and 59% of the Quantcast top-million web sites [3]. While component reuse in the venerable desktop software model typically involves libraries, the reusable components in web applications are not limited to just JavaScript library code—they further include network-accessible content and services.

The resulting model is one in which web developers cobble together multiple JavaScript libraries, web-based content, and web-based services written and operated by various parties (who in turn may integrate more of these resources) and build the required application-specific functionality atop them. Unfortunately, some of the many contributors to the tangle of JavaScript comprising an application may not have the user's best interest at heart. The wealth of sensitive data processed in today's web applications (*e.g.,* email, bank statements, health records, passwords, *etc.*) is an attractive target. Miscreants may stealthily craft malicious JavaScript that, when incorporated into an application by an unwitting developer, violates the user's privacy by leaking sensitive information.

Two goals for web applications emerge from the prior discussion: *flexibility* for the application developer (*i.e.,* enabling the building of applications with rich functionality, composable from potentially disparate pieces hosted by different sites); and *privacy* for the user (*i.e.,* to ensure that the user's sensitive data cannot be leaked from applications to unauthorized parties). These two goals are hardly new: Wang *et al.* articulated similar ones, and proposed new browser primitives to improve isolation within *mashups,* including discretionary access control (DAC) for inter-frame communication [41]. Indeed, today's browsers incorporate similar mechanisms in the guises of HTML5's iframe sandbox and postMessage API [47]. And the *Same-Origin Policy* (SOP, reviewed in Section 2.1) prevents JavaScript hosted by one principal from reading content hosted by another.

Unfortunately, in the status-quo web browser security architecture, one must often sacrifice privacy to achieve flexibility, and vice-versa. The central reason that flexibility and privacy are at odds in the status quo is that the mechanisms today's browsers rely on for providing privacy—the SOP, Content Security Policy (CSP) [42], and Cross-Origin Resource Sharing (CORS) [45]—are all forms of discretionary access control. DAC has the brittle character of either denying or granting untrusted code (*e.g.,* a library written by a third party) access to data. In the former case, the untrusted JavaScript might *need* the sensitive data to implement the desired application functionality—hence, denying access prioritizes privacy over flexibility. In the latter, DAC exercises no control over what the untrusted code does with the sensitive data—and thus prioritizes flexibility over privacy. DAC is an essential tool in the privacy arsenal, but *does not fit cases where one runs untrusted code on sensitive input,* which are the norm for web applications, given their multi-contributor nature.

In practice, web developers turn their backs on privacy in favor of flexibility because the browser doesn't offer

[*] Work partly conducted while at Mozilla.

[†] Work partly conducted while at Stanford.

primitives that let them opt for both. For example, a developer may want to include untrusted JavaScript from another origin in his application. All-or-nothing DAC leads the developer to include the untrusted library with a `script` tag, which effectively bypasses the SOP, interpolating untrusted code into the enclosing page and granting it unfettered access to the enclosing page's origin's content.[1] And when a developer of a mashup that integrates content from *other* origins finds that the SOP forbids his application from retrieving data from them, he designs his mashup to require that the user provide the mashup her login credentials for the sites at the two other origins [2]—the epitome of "functionality over privacy."

In this paper, we present COWL (Confinement with Origin Web Labels), a mandatory access control (MAC) system that confines untrusted JavaScript in web browsers. COWL allows untrusted code to compute over sensitive data and display results to the user, but prohibits the untrusted code from exfiltrating sensitive data (*e.g.,* by sending it to an untrusted remote origin). It thus allows web developers to opt for *both* flexibility and privacy.

We consider four motivating example web applications—a password strength-checker, an application that imports the (untrusted) jQuery library, an encrypted cloud-based document editor, and a third-party mashup, none of which can be implemented in a way that preserves the user's privacy in the status-quo web security architecture. These examples drive the design requirements for COWL, particularly MAC with *symmetric and hierarchical confinement* that supports *delegation.* Symmetric confinement allows *mutually* distrusting principals each to pass sensitive data to the other, and confine the other's use of the passed sensitive data. Hierarchical confinement allows any developer to confine code she does not trust, and confinement to be nested to arbitrary depths. And delegation allows a developer explicitly to confer the privileges of one execution context on a separate execution context. No prior browser security architecture offers this combination of properties.

We demonstrate COWL's applicability by implementing secure versions of the four motivating applications with it. Our contributions include:

- ▶ We characterize the shared needs of four case-study web applications (Section 2.2) for which today's browser security architecture cannot provide privacy.
- ▶ We describe the design of the COWL label-based MAC system for web browsers (Section 3), which meets the requirements of the four case-study web applications.
- ▶ We describe designs of the four case-study web applications atop COWL (Section 4).
- ▶ We describe implementations of COWL (Section 5) for the Firefox and Chromium open-source browsers; our evaluation (Section 6) illustrates that COWL incurs minimal performance overhead over the respective baseline browsers.

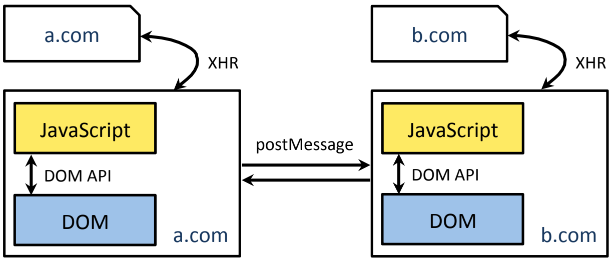


Figure 1: Simplified browser architecture.

2 Background, Examples, & Goals

A single top-level web page often incorporates multiple scripts written by different authors.[2] Ideally, the browser should protect the user's sensitive data from unauthorized disclosure, yet afford page developers the greatest possible flexibility to construct featureful applications that reuse functionality implemented in scripts provided by (potentially untrusted) third parties. To make concrete the diversity of potential trust relationships between scripts' authors and the many ways page developers structure amalgamations of scripts, we describe several example web applications, none of which can be implemented with strong privacy for the user in today's web browsers. These examples illustrate key requirements for the design of a flexible browser confinement mechanism. Before describing these examples, however, we offer a brief refresher on status-quo browser privacy polices.

2.1 Browser Privacy Policies

Browsing contexts Figure 1 depicts the basic building blocks of the current web security architecture. A *browsing context* (*e.g.,* a page or frame) encapsulates presentable content and a JavaScript execution environment (heap and code) that interacts with content through the *Document Object Model (DOM)* [47]. Browsing contexts may be nested (*e.g.,* by using iframes). They also may read and write persistent storage (*e.g.,* cookies), issue network requests (either implicitly in page content that references a URL retrieved over the network, or explicitly in JavaScript, using the `XMLHttpRequest` (XHR) constructor), and communicate with other contexts (IPC-style via `postMessage`, or, in certain cases, by sharing DOM objects). Some contexts such as Web Workers [44] run JavaScript but do not instantiate a DOM. We use the terms *context* and *compartment* interchangeably to refer to both browsing contexts and workers, except when the more precise meaning is relevant.

Origins and the Same-Origin Policy Since different authors may contribute components within a page, today's

[1] Indeed, jQuery *requires* such access to the enclosing page's content!

[2] Throughout we use "web page" and "web application" interchangeably, and "JavaScript code" and "script" interchangeably.

status quo browsers impose a security policy on interactions among components. Policies are expressed in terms of *origins*. An origin is a source of authority encoded by the protocol (*e.g.,* `https`), domain name (*e.g.,* `fb.com`), and port (*e.g.,* `443`) of a resource URL. For brevity, we elide the protocol and port from URLs throughout.

The same-origin policy specifies that an origin's resources should be readable only by content from the same origin [7, 38, 52]. Browsers ensure that code executing in an `a.com` context can only inspect the DOM and cookies of another context if they share the same origin, *i.e.,* `a.com`. Similarly, such code can only inspect the response to a network request (performed with XHR) if the remote host's origin is `a.com`.

The SOP does not, however, prevent code from *disclosing* data to foreign origins. For example, code executing in an `a.com` context can trivially disclose data to `b.com` by using XHR to perform a network request; the SOP prevents the code from inspecting responses to such cross-origin XHR requests, but does not impose any restrictions on sending such requests. Similarly, code can exfiltrate data by encoding it in the path of a URL whose origin is `b.com`, and setting the `src` property of an `img` element to this URL.

Content Security Policy (CSP) Modern browsers allow the developer to protect a user's privacy by specifying a CSP that limits the communication of a page—*i.e.,* that disallows certain communication ordinarily permitted by the SOP. Developers may set individual CSP directives to restrict the origins to which a context may issue requests of specific types (for images or scripts, XHR destinations, *etc.*) [42]. However, CSP policies suffer from two limitations. They are *static:* they cannot change during a page's lifetime (*e.g.,* a page may not drop the privilege to communicate with untrusted origins before reading potentially sensitive data). And they are *inaccessible:* JavaScript code cannot inspect the CSP of its enclosing context or some other context, *e.g.,* when determining whether to share sensitive data with that other context.

postMessage and Cross-Origin Resource Sharing (CORS) As illustrated in Figure 1, the HTML5 `postMessage` API [43] enables cross-origin communication in IPC-like fashion within the browser. To prevent unintended leaks [8], a sender always specifies the origin of the intended recipient; only a context with that origin may read the message.

CORS [45] goes a step further and allows controlled cross-origin communication between a browsing context of one origin and a remote server with a different origin. Under CORS, a server may include a header on returned content that explicitly whitelists other origin(s) allowed to read the response.

Note that both `postMessage`'s target origin and CORS are purely discretionary in nature: they allow static selection of which cross-origin communication is allowed and which denied, but enforce no confinement on a receiving compartment of differing origin. Thus, in the status-quo web security architecture, a privacy-conscious developer should only send sensitive data to a compartment of differing origin if she completely trusts that origin.

2.2 Motivating Examples

Having reviewed the building blocks of security policies in status-quo web browsers, we now turn to examples of web applications for which strong privacy is not achievable today. These examples illuminate key design requirements for the COWL confinement system.

Password Strength Checker Given users' propensity for choosing poor (*i.e.,* easily guessable) passwords, many web sites today incorporate functionality to check the strength of a password selected by a user and offer the user feedback (*e.g.,* "too weak; choose another," "strong," *etc.*). Suppose a developer at Facebook (origin `fb.com`) wishes to re-use password-checking functionality provided in a JavaScript library by a third party, say, from origin `sketchy.ru`. If the developer at `fb.com` simply includes the third party's code in a `script` tag referencing a resource at `sketchy.ru`, then the referenced script will have unfettered access to both the user's password (provided by the Facebook page, which the library *must* see to do its job) and to write to the network via XHR. This simple state of affairs is emblematic of the ease with which naïve web developers can introduce leaks of sensitive data in applications.

A more skilled web developer could today host the checker script on her *own* server and have that server specify a CSP policy for the page. Unfortunately, a CSP policy that disallows scripts within the page from initiating XHRs to any other origins is *too inflexible,* in that it precludes useful operations by the checker script, *e.g.,* retrieving an updated set of regular expressions describing weak passwords from a remote server (essentially, "updating" the checker's functionality). Doing so requires communicating with a remote origin. Yet a CSP policy that permits such communication, even with the top-level page's same origin, is *too permissive:* a malicious script could potentially carry out a *self-exfiltration attack* and write the password to a public part of the trusted server [11, 50].

This trade-off between flexibility and privacy, while inherent to CSP, need not be fundamental to the web model. The key insight is that it is entirely safe and useful for an untrusted script to communicate with remote origins *before* it reads sensitive data. We note, then, the requirement of a confinement mechanism that allows code in a compartment to communicate with the network *until it has been exposed to sensitive data.* MAC-based confinement meets this requirement.

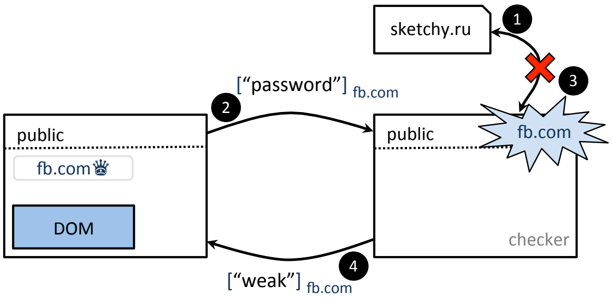


Figure 2: Third-party password checker architecture under COWL.

Figure 2 shows how such a design might look. In this and subsequent examples, rectangular frames denote compartments, arrows denote communication (either between a compartment and the network, or IPC-style between compartments), and events during execution are numbered sequentially in time. As we have proposed previously [49], compartments may be *labeled* (Section 3.1) with the origins to whose sensitive data they have been exposed. A compartment that has not yet observed sensitive data is denoted `public`; however, when it wishes to incorporate sensitive data, the compartment *raises* its label (at the cost of being more restricted in where it can write). We illustrate the raising of a label with a "flash" connoting the sensitivity of data being integrated. A compartment's *privilege* (Section 3.3), which specifies the origins for which a script executing in that compartment is trusted, is indicated by a crown. Here, a top-level page at `fb.com` encapsulates a password-checker script from a third-party origin in a new compartment. The label of the new compartment is initially `public`. First, in step (1), the checker script is free to download updated regular expressions from an arbitrary remote origin. In step (2), the top-level page sends the user's password to the checker script's worker using `postMessage`; the password is *labeled* `fb.com` to indicate that the data is sensitive to this origin (Section 3.2). In step (3) the checker raises its label to reflect that the context is about to be exposed to sensitive data from `fb.com` and inspects the password. When the label is raised, COWL atomically denies the context further access to the network in step (3).[3] However, the checker script is free to compute the result, which it then returns via `postMessage` to the top-level page in step (4); the result carries the label `fb.com` to reflect that the sender may be sending data derived from sensitive data owned by `fb.com`. Since the top-level page has the `fb.com` privilege, it can simply read the data (without raising its label).

Encrypted Document Editor Today's web applications, such as in-browser document editors backed by cloud-based storage (*e.g.,* Google Docs), typically require the user to trust the app developer/cloud-based storage provider (often the same principal under the SOP) with the data in her documents. That is, the provider's server observes the user's data in cleartext. Suppose an organization wished to use an in-browser document editor but did *not* want to reveal its users' document data to the editor provider's server. How might the provider offer a privacy-preserving editor app that would satisfy the needs of such a privacy-conscious organization? One promising approach might be for the "customer" privacy-sensitive organization to implement a trusted document encryption service hosted at its own origin, distinct from that which hosts the editor app. The editor app could allow the user to specify a JavaScript "plugin" library she trusts to perform cryptography correctly. In this design, one origin serves the JavaScript code for the editor app (say, `gdocs.com`) and a different origin serves the JavaScript code for the cryptography library (say, `eff.org`). Note that these two origins may be *mutually distrusting.* `gdocs.com`'s script must pass the document's cleartext to a script from `eff.org` for encryption, but would like to confine the execution of the encryption script so that it cannot exfiltrate the document to any origin *other* than `gdocs.com`. Similarly, `eff.org`'s cryptography library may not trust `gdocs.com` with the cleartext document—it would like to confine `gdocs.com`'s editor to prevent exfiltration of the cleartext document to `gdocs.com` (or to any other origin). This simple use case highlights the need for *symmetric confinement:* when two mutually distrusting scripts from different origins communicate, *each must be able to confine the other's further use of data it provides.*

Third-Party Mashup Some of the most useful web applications are *mashups*; these applications integrate and compute over data hosted by multiple origins. For example, consider an application that reconciles a user's Amazon purchases (the data for which are hosted by `amazon.com`) against a user's bank statement (the data for which are hosted by `chase.com`). The user may well deem both these categories of data sensitive and will furthermore not want data from Amazon to be exposed to her bank or vice-versa, nor to any other remote party. Today, if one of the two providers implements the mashup, its application code must bypass the SOP to allow sharing of data across origin boundaries, *e.g.,* by communicating between iframes with `postMessage` or setting a permissive CORS policy. This approach forfeits privacy: one origin sends sensitive data to the other, after which the receiving origin may exfiltrate that sensitive data at will. Alternatively, a third-party developer may wish to implement and offer this mashup application. Users of such a *third-party mashup* give up their privacy, usually by simply

[3] For clarity, we use `fb.com` as the label on the data. This label still allows the checker to send XHR requests to `fb.com`; to ensure that the checker cannot communicate with *any* origin, COWL provides fresh origins (see Section 3.3).

handing off credentials, as again today's browser enforces no policy that confines the sensitive data the mashup's code observes within the browser. To enable third-party mashups that do not sacrifice the user's privacy, we note again the need for an untrusted script to be able to issue requests to multiple remote origins (*e.g.,* `amazon.com` and `chase.com`), but to lose the privilege to communicate over the network once it has read the responses from those origins. Here, too, MAC-based confinement addresses the shortcomings of DAC.

Untrusted Third-Party Library Web application developers today make extensive use of third-party libraries like jQuery. Simply importing a library into a page provides no isolation whatsoever between the untrusted third-party code and any sensitive data within the page. Developers of applications that process sensitive data want the convenience of reusing popular libraries. But such reuse risks exfiltration of sensitive data by these untrusted libraries. Note that because jQuery requires access to the content of the entire page that uses it, we cannot isolate jQuery in a separate compartment from the parent's, as we did for the password-checker example. Instead, we observe that jQuery demands a design that is a mirror image of that for confining the password checker: we place the *trusted* code for a page in a separate compartment and deem the rest of the page (including the untrusted jQuery code) as untrusted. The trusted code can then communicate with remote origins and inject sensitive data into the untrusted page, but the untrusted page (including jQuery) cannot communicate with remote origins (and thus cannot exfiltrate sensitive data within the untrusted page). This refactoring highlights the need for a confinement system that supports *delegation* and *dropping privilege*: a page should be able to create a compartment, confer its privileges to communicate with remote origins on that compartment, and then give these privileges up.

We note further that any *library* author may wish to reuse functionality from another untrusted library. Accordingly, to allow the broadest reuse of code, the browser should support *hierarchical confinement*—the primitives for confining untrusted code should allow not only a single level of confinement (one trusted context confining one untrusted context), but arbitrarily many levels of confinement (one trusted context confining an untrusted one, that in turn confines a further untrusted one, *etc.*).

2.3 Design Goals

We have briefly introduced four motivating web applications that achieve rich functionality by combining code from one or more untrusted parties. The privacy challenges that arise in such applications are unfortunately unaddressed by status-quo browser security policies, such as the SOP. These applications clearly illustrate the need for robust yet flexible confinement for untrusted code in browsers. To summarize, these applications would appear to be well served by a system that:

- Applies mandatory access control (MAC);
- Is *symmetric, i.e.,* it permits two principals to *mutually* distrust one another, and each prevent the other from exfiltrating its data;
- Is *hierarchical, i.e.,* it permits principal *A* to confine code from principal *B* that processes *A*'s data, while principal *B* can independently confine code from principal *C* that processes *B*'s data, *etc.*
- Supports *delegation* and *dropping privilege, i.e.,* it permits a script running in a compartment with the privilege to communicate with some set of origins to confer those privileges on another compartment, then relinquish those privileges itself.

In the next section, we describe COWL, a new confinement system that satisfies these design goals.

3 The COWL Confinement System

The COWL confinement system extends the browser security model while leaving the browser fully compatible with today's "legacy" web applications.[4] Under COWL, the browser treats a page exactly like a legacy browser does unless the page executes a COWL API operation, at which point the browser records that page as running in *confinement mode,* and all further operations by that page are subject to confinement by COWL. COWL augments today's web browser with three primitives, all of which appear in the simple password-checker application example in Figure 2.

Labeled browsing contexts enforce MAC-based confinement of JavaScript at the granularity of a context (*e.g.,* a worker or iframe). The rectangular frames in Figure 2 are labeled contexts. As contexts may be nested, labeled browsing contexts allow hierarchical confinement, whose importance for supporting nesting of untrusted libraries we discussed in Section 2.2.

When one browsing context sends sensitive information to another, a sending context can use *labeled communication* to confine the potentially untrusted code receiving the information. This enables symmetric confinement, whose importance in building applications that compose mutually distrusting scripts we articulated in Section 2.2. In Figure 2, the arrows between compartments indicate labeled communication, where a subscript on the communicated data denotes the data's label.

COWL may grant a labeled browsing context one or more *privileges,* each with respect to an origin, and each of which reflects trust that the scripts executing within

[4] In prior work, we described how confinement can subsume today's browser security primitives, and advocated replacing them entirely with a clean-slate, confinement-based model [49]. In this paper, we instead prioritize incremental deployability, which requires coexistence alongside the status quo model.

that context will not violate the secrecy and integrity of that origin's data, *e.g.,* because the browser retrieved them from that origin. A privilege authorizes scripts within a context to execute certain operations, such as *declassification* and *delegation*, whose abuse would permit the release of sensitive information to unauthorized parties. In COWL, we express privilege in terms of origins. The crown icon in the left compartment in Figure 2 denotes that this compartment may execute privileged operations on data labeled with the origin `fb.com`—more succinctly, that the compartment holds the privilege for `fb.com`. The compartment uses that privilege to remain unconfined by declassifying the checker response labeled `fb.com`.

We now describe these three constructs in greater detail.

3.1 Labeled Browsing Contexts

A COWL application consists of multiple labeled contexts. Labeled contexts extend today's browser contexts, used to isolate iframes, pages, *etc.*, with MAC *labels*. A context's label specifies the security policy for all data within the context, which COWL enforces by restricting the flow of information to and from other contexts and servers.

As we have proposed previously [33, 49], a label is a pair of boolean formulas over origins: a *secrecy* formula specifying which origins may read a context's data, and an *integrity* formula specifying which origins may write it. For example, only Amazon or Chase may read data labeled ⟨`amazon.com` ∨ `chase.com`, `amazon.com`⟩, and only Amazon may modify it.[5] Amazon could assign this label to its order history page to allow a Chase-hosted mashup to read the user's purchases. On the other hand, after a third-party mashup hosted by `mint.com` (as described in Section 2.2) reads *both* the user's Chase bank statement data *and* Amazon purchase data, the label on data produced by the third-party mashup will be ⟨`amazon.com` ∧ `chase.com`, `mint.com`⟩. This secrecy label component specifies that the data may be sensitive to both parties, and without both their consent (see Section 3.3), it should only be read by the user; the integrity label component, on the other hand, permits only code hosted by Mint to modify the resulting data.

COWL enforces label policies in a MAC fashion by only allowing a context to communicate with other contexts or servers whose labels are at least as restricting. (A server's "label" is simply its origin.) Intuitively, when a context wishes to send a message, the target must not allow additional origins to read the data (preserving secrecy). Dually, the source context must not be writable by origins not otherwise trusted by the target. That is, the source must be at least as trustworthy as the target. We say that such a target label "subsumes" the source label. For example, a context labeled ⟨`amazon.com`, `mint.com`⟩ can send messages to one labeled ⟨`amazon.com` ∧ `chase.com`, `mint.com`⟩, since the latter is trusted to preserve the privacy of `amazon.com` (and `chase.com`). However, communication in the reverse direction is not possible since it may violate the privacy of `chase.com`. In the rest of this paper, we limit our discussion to secrecy and only comment on integrity where relevant; we refer the interested reader to [33] for a full description of the label model.

A context can freely *raise* its label, *i.e.,* change its label to any label that is more restricting, in order to receive a message from an otherwise prohibited context. Of course, in raising its label to read more sensitive data from another context, the context also becomes more restricted in where it can write. For example, a Mint context labeled ⟨`amazon.com`⟩ can raise its label to ⟨`amazon.com` ∧ `chase.com`⟩ to read bank statements, but only at the cost of giving up its ability to communicate with Amazon (or, for that matter, any other) servers. When creating a new context, code can impose an upper bound on the context's label to ensure that untrusted code cannot raise its label and read data above this *clearance*. This notion of clearance is well established [14, 17, 34, 35, 51]; we discuss its relevance to covert channels in Section 7.

As noted, COWL allows a labeled context to create additional labeled contexts, much as today's browsing contexts can create sub-compartments in the form of iframes, workers, *etc.* This functionality is crucial for compartmentalizing a system hierarchically, where the developer places code of different degrees of trustworthiness in separate contexts. For example, in the password checker example in Section 2.2, we create a child context in which we execute the untrusted checker script. Importantly, however, code should not be able to leak information by laundering data through a newly created context. Hence, a newly created context implicitly inherits the current label of its parent. Alternatively, when creating a child, the parent may specify an initial current label for the child that is *more* restrictive than the parent's, to confine the child further. Top-level contexts (*i.e.,* pages) are assigned a default label of `public`, to ensure compatibility with pages written for the legacy SOP. Such browsing contexts can be restricted by setting a `COWL-label` HTTP response header, which dictates the minimal document label the browser must enforce on the associated content.

COWL applications can create two types of context. First, an application can create standard (but labeled) contexts in the form of pages, iframes, workers, *etc.* Indeed, it may do so because a COWL application is merely a regular web application that additionally uses the COWL API. It thus is confined by MAC, in addition to today's web security policies. Note that to enforce MAC, COWL must mediate all pre-existing communication channels—even

[5] ∨ and ∧ denote disjunction and conjunction. A comma separates the secrecy and integrity formulas.

subtle and implicit channels, such as content loading—according to contexts' labels. We describe how COWL does so in Section 5.

Second, a COWL application can create labeled contexts in the form of *lightweight labeled workers (LWorkers)*. Like normal workers [44], the API exposed to LWorkers is minimal; it consists only of constructs for communicating with the parent, the XHR constructor, and the COWL API. Unlike normal workers, which execute in separate threads, an LWorker executes in the same thread as its parent, sharing its event loop. This sharing has the added benefit of allowing the parent to give the child (labeled) access to its DOM, any access to which is treated as both a read and a write, *i.e.,* bidirectional communication. Our third-party library example uses such a *DOM worker* to isolate the trusted application code, which requires access to the DOM, from the untrusted jQuery library. In general, LWorkers—especially when given DOM access—simplify the isolation and confinement of scripts (*e.g.,* the password strength checker) that would otherwise run in a shared context, as when loaded with `script` tags.

3.2 Labeled Communication

Since COWL enforces a label check whenever a context sends a message, the design described thus far is already symmetric: a source context can confine a target context by raising its label (or a child context's label) and thereafter send the desired message. To read this message, the target context must confine itself by raising its label accordingly. These semantics can make interactions between contexts cumbersome, however. For example, a sending context may wish to communicate with multiple contexts, and need to confine those target contexts with different labels, or even confine the same target context with different labels for different messages. And a receiving context may need unfettered communication with one or more origins for a time before confining itself by raising its label to receive a message. In the password-checker example application, the untrusted checker script at the right of Figure 2 exhibits exactly this latter behavior: it needs to communicate with untrusted remote origin `sketchy.ru` before reading the password labeled `fb.com`.

Labeled Blob Messages (Intra-Browser) To simplify communication with confinement, we introduce the *labeled Blob,* which binds together the payload of an individual inter-context message with the label protecting it. The payload takes the form of a serialized immutable object of type Blob [47]. Encapsulating the label with the message avoids the cumbersome label raises heretofore necessary in both sending and receiving contexts before a message may even be sent or received. Instead, COWL allows the developer sending a message from a context to specify the label to be attached to a labeled Blob; any label as or more restrictive than the sending context's current label may be specified (modulo its clearance). While the receiving context may receive a labeled Blob with no immediate effect on the origins with which it can communicate, it may only inspect the label, not the payload.[6] Only after raising its label as needed may the receiving context read the payload.

Labeled Blobs simplify building applications that incorporate distrust among contexts. Not only can a sender impose confinement on a receiver simply by labeling a message; a receiver can delay inspecting a sensitive message until it has completed communication with untrusted origins (as does the checker script in Figure 2). They also ease the implementation of integrity in applications, as they allow a context that is not trusted to modify content in some other context to serve as a passive conduit for a message from a third context that *is* so trusted.

Labeled XHR Messages (Browser–Server) Thus far we have focused on confinement as it arises when two browser contexts communicate. Confinement is of use in browser-server communication, too. As noted in Section 3.1, COWL only allows a context to communicate with a server (whether with XHR, retrieving an image, or otherwise) when the server's origin subsumes the context's label. Upon receiving a request, a COWL-aware web server may also wish to know the current label of the context that initiated it. For this reason, COWL attaches the current label to every request the browser sends to a server.[7] As also noted in Section 3.1, a COWL-aware web server may elect to label a response it sends the client by including a `COWL-label` header on it. In such cases, the COWL-aware browser will only allow the receiving context to read the XHR response if its current label subsumes that on the response.

Here, again, a context that receives labeled data—in this case from a server—may wish to defer raising its label until it has completed communication with other remote origins. To give a context this freedom, COWL supports *labeled XHR* communication. When a script invokes COWL's labeled XHR constructor, COWL delivers the response to the initiating script as a labeled Blob. Just as with labeled Blob intra-browser IPC, the script is then free to delay raising its label to read the payload of the response—and delay being confined—until after it has completed its other remote communication. For example, in the third-party mashup example, Mint only confines itself once it has received all necessary (labeled) responses from both Amazon and Chase. At this point it processes the data and displays results to the user, but it can no longer send requests since doing so may leak

[6] The label itself cannot leak information—COWL still ensures that the target context's label is at least as restricting as that of the source.

[7] COWL also attaches the current privilege; see Section 3.3.

information.[8]

3.3 Privileges

While confinement handily enforces secrecy, there are occasions when an application must eschew confinement in order to achieve its goals, and yet can uphold secrecy while doing so. For example, a context may be confined with respect to some origin (say, `a.com`) as a result of having received data from that origin, but may need to send an encrypted version of that data to a third-party origin. Doing so does not disclose sensitive data, but COWL would normally prohibit such an operation. In such situations, how can a context *declassify* data, and thus be permitted to send to an arbitrary recipient, or avoid the recipient's being confined?

COWL's *privilege* primitive enables safe declassification. A context may hold one or more privileges, each with respect to some origin. Possession of a privilege for an origin by a context denotes trust that the scripts that execute within that context will not compromise the secrecy of data from that origin. Where might such trust come from (and hence how are privileges granted)? Under the SOP, when a browser retrieves a page from `a.com`, any script within the context for the page is trusted not to violate the secrecy of `a.com`'s data, as these scripts are deemed to be executing on behalf of `a.com`. COWL makes the analogous assumption by granting the privilege for `a.com` to the context that retrieves a page from `a.com`: scripts executing in that context are similarly deemed to be executing on behalf of `a.com`, and thus are trusted not to leak `a.com`'s data to unauthorized parties—even though they can declassify data. Only the COWL runtime can create a new privilege for a valid remote origin upon retrieval of a page from that origin; a script cannot synthesize a privilege for a valid remote origin.

To illustrate the role of privileges in declassification, consider the encrypted Google Docs example application. In the implementation of this application atop COWL, code executing on behalf of `eff.org` (*i.e.,* in a compartment holding the `eff.org` privilege) with a current label $\langle \texttt{eff.org} \wedge \texttt{gdoc.com} \rangle$ is permitted to send messages to a context labeled $\langle \texttt{gdoc.com} \rangle$. Without the `eff.org` privilege, this flow would not be allowed, as it may leak the EFF's information to Google.

Similarly, code can declassify information when unlabeling messages. Consider now the password checker example application. The left context in Figure 2 leverages its `fb.com` privilege to declassify the password strength result, which is labeled with its origin, to avoid (unecessarily) raising its label to `fb.com`.

COWL generally exercises privileges *implicitly:* if a context holds a privilege, code executing in that context will, with the exception of sending a message, always attempt to use it.[9] COWL, however, lets code control the use of privileges by allowing code to get and set the underlying context's privileges. Code can drop privileges by setting its context's privileges to **`null`**. Dropping privileges is of practical use in confining closely coupled untrusted libraries like jQuery. Setting privileges, on the other hand, increases the trust placed in a context by authorizing it act on behalf of origins. This is especially useful since COWL allows one context to *delegate* its privileges (or a subset of them) to another; this functionality is also instrumental in confining untrusted libraries like jQuery. Finally, COWL also allows a context to create privileges for *fresh* origins, *i.e.,* unique origins that do not have a real protocol (and thus do not map to real servers). These fresh origins are primarily used to *completely* confine a context: the sender can label messages with such an origin, which upon inspection will raise the receiver's label to this "fake" origin, thereby ensuring that it cannot communicate except with the parent (which holds the fresh origin's privilege).

4 APPLICATIONS

In Section 2.2, we characterized four applications and explained why the status-quo web architecture cannot accommodate them satisfactorily. We then described the COWL system's new browser primitives. We now close the loop by demonstrating how to build the aforementioned applications with the COWL primitives.

Encrypted Document Editor The key feature needed by an encrypted document editor is symmetric confinement, where two mutually distrusting scripts can each confine the other's use of data they send one another. Asymmetrically conferring COWL privileges on the distrusting components is the key to realizing this application.

Figure 3 depicts the architecture for an encrypted document editor. The editor has three components: a component which has the user's Google Docs credentials and communicates with the server (`gdoc.com`), the editor proper (also `gdoc.com`), and the component that performs encryption (`eff.org`). COWL provides privacy as follows: if `eff.org` is honest, then COWL ensures that the cleartext of the user's document is not leaked to any origin. If only `gdoc.com` is honest, then `gdoc.com` may be able to recover cleartext (*e.g.,* the encryptor may have used the null "cipher"), but the encryptor should not be able to exfiltrate the cleartext to anyone else.

How does execution of the encrypted document editor proceed? Initially, `gdoc.com` downloads (1) the en-

[8] To continuously process data in "streaming" fashion, one may partition the application into contexts that poll Amazon and Chase's servers for new data and pass labeled responses to the confined context that processes the payloads of the responses.

[9] While the alternative approach of explicit exercise of privileges (*e.g.,* when registering an `onmessage` handler) may be safer [23, 34, 51], we find it a poor fit with existing asynchronous web APIs.

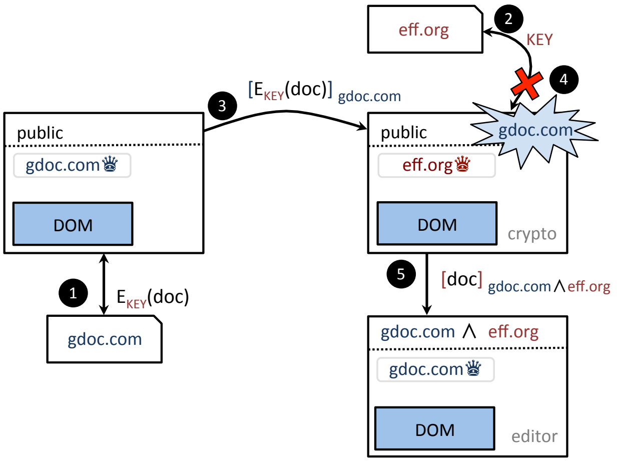


Figure 3: Encrypted document editor architecture.

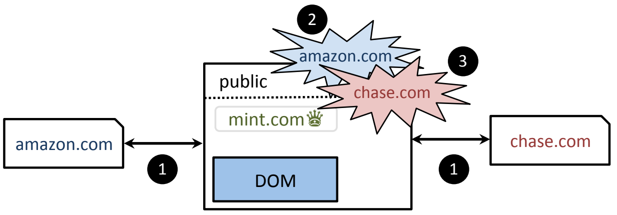


Figure 4: Third-party mashup under COWL.

crypted document from Google's servers. As the document is encrypted, it opens an iframe to `eff.org`, with initial label `public` so it can communicate with the `eff.org` server and download the private key (2) which will be used to decrypt the document. Next, it sends the encrypted document as a labeled Blob, with the label ⟨`gdoc.com`⟩ (3); the iframe unlabels the Blob and raises its label (4) so it can decrypt the document. Finally, the iframe passes the decrypted document (labeled as ⟨`gdoc.com` ∧ `eff.org`⟩) to the iframe (5) implementing the editor proper.

To save the document, these steps proceed in reverse: the editor sends a decrypted document to the encryptor (5), which encrypts it with the private key. Next, the critical step occurs: the encryptor exercises its privileges to send a labeled blob of the encrypted document which is *only* labeled ⟨`gdoc.com`⟩ (3). Since the encryptor is the only compartment with the `eff.org` privilege, all documents must pass through it for encryption before being sent elsewhere; conversely, it itself cannot exfiltrate any data, as it is confined by `gdoc.com` in its label.

We have implemented a password manager atop COWL that lets users safely store passwords on third-party web-accessible storage. We elide its detailed design in the interest of brevity, and note only that it operates similarly to the encrypted document editor.

Third-Party Mashup Labeled XHR as composed with CORS is central to COWL's support for third-party mashups. Today's CORS policies are DAC-only, such that a server must either allow another origin to read its data and fully trust that origin not to disclose the data, or deny the other origin access to the data altogether. Under COWL, however, a server could CORS-whitelist a foreign origin to permit that origin to read its data, and by setting a label on its response, be safe in the knowledge that COWL would appropriately confine the foreign origin's scripts in the browser.

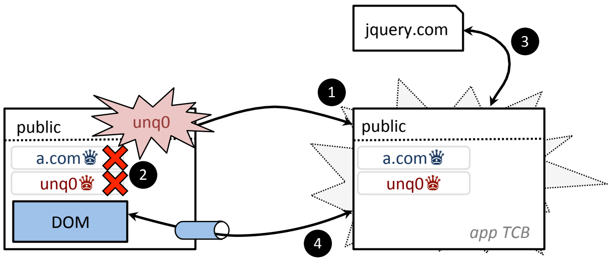


Figure 5: Privilege separation and library confinement.

Figure 4 depicts an application that reconciles a user's Amazon purchases and bank statement. Here, Chase and Amazon respectively expose authenticated read-only APIs for bank statements and purchase histories that whitelist known applications' origins, such as `mint.com`, but set MAC labels on responses.[10] As discussed in Section 7, with MAC in place, COWL allows users to otherwise augment CORS by whitelisting foreign origins on a per-origin basis. The mashup makes requests to both web sites using labeled XHR (1) to receive the bank statement and purchase history as labeled Blobs. Once all of the information is received, the mashup unlabels the responses and raises its context's label accordingly (2–3); doing so restricts communication to the web at large.

Note that in contrast to when solely using CORS, by setting MAC labels on responses, Chase and Amazon need not trust Mint to write bug-free code—COWL confines the Mint code to ensure that it cannot arbitrarily leak sensitive data. As we discuss in Section 7, however, a malicious Mint application could potentially leak data through covert channels. We emphasize that COWL nevertheless offers a significant improvement over the status quo, in which, *e.g.,* users give their login credentials to Mint, and thus not only trust Mint to keep their bank statements confidential, but also not to steal their funds!

Untrusted Third-Party Library COWL can confine tightly coupled untrusted third-party libraries like jQuery by delegating privileges to a trusted context and subsequently dropping them from the main page. In doing so, COWL completely confines the main page, and ensures that it can only communicate with the trusted and unconfined context. Here, the main page may start out with sensitive data in context, or alternatively, receive it from the trusted compartment.

[10] On authentication: note that when the browser sends any XHR (labeled or not) from a foreign origin to origin `chase.com`, it still includes any cookies cached for `chase.com` in the request.

```
interface Label :
  Label Label(String)
  Label and(String or Label)
  Label or(String or Label)
  bool subsumes(Label [,Privilege])
```

```
interface Privilege :
  Privilege FreshPrivilege()
  Privilege combine(Privilege)
  readonly attribute Label asLabel
```

(a) Labels and privileges.

```
interface LabeledBlob :
  readonly attribute Label label
  readonly attribute Blob blob
```

(b) Labeled Blobs.

```
interface COWL :
  static void enable()
  static attribute Label label
  static attribute Label clearance
  static attribute Privilege privilege
```

```
interface LWorker :
  LWorker LWorker(String, Label
                  [, Privilege, object])
  postMessage(object)
  attribute EventHandler onmessage
```

(c) Labeled compartments.

Figure 6: COWL programming interface in simplified WebIDL.

Figure 5 shows how to use COWL to confine the untrusted jQuery library referenced by a web page. The goal is to establish a separate DOM worker with the `a.com` privilege, while the main browsing context runs jQuery in confined fashion—without privileges or the ability to talk to the network. Initially the main browsing context holds the `a.com` privilege. The page generates a fresh origin `unq0` and spawns a DOM worker (1), delegating it both privileges. The main context then drops its privileges and raises its label to ⟨`unq0`⟩ (2). Finally, the trusted worker downloads jQuery (3) and injects the script content into the main context's DOM (4). When the library is loaded, the main context becomes untrusted, but also fully confined. As the trusted DOM worker holds both privileges, it can freely modify the DOM of the main context, as well as communicate with the wider web. One may view this DOM worker as a *firewall* between the page proper (with the untrusted library) and the rest of the world.

5 IMPLEMENTATION

We implemented COWL in Firefox 31.0a1 and Chromium 31.0.1612.0. Because COWL operates at a context granularity, it admits an implementation as a new DOM-level API for the Gecko and Blink layout engines, without any changes to the browsers' JavaScript engines. Figure 6 shows the core parts of this API. We focus on the Firefox implementation and only describe the Chromium one where the two diverge non-trivially.

Channel	Mechanism
`postMessage`	Cross-compartment wrappers[11]
DOM window properties	Cross-compartment wrappers
Content loading	CSP
XHR	CSP + DOM interposition
Browser storage	SOP + CSP (sandbox)
Other (*e.g.,* iframe height)	DOM interposition

Table 1: Confining code from exfiltrating data using existing browser mechanisms.

5.1 Labeled Browsing Contexts

Gecko's existing isolation model relies on JavaScript compartments, *i.e.,* disjoint JavaScript heaps, both for efficient garbage collection and security isolation [40]. To achieve isolation, Gecko performs all cross-compartment communication (*e.g.,* `postMessage` between iframes) through *wrappers* that implement the object-capability *membrane* pattern [21, 22]; membranes enable sound reasoning about "border crossing" between compartments. Wrappers ensure that an object in one compartment can never directly reference another object in a different compartment. Wrappers also include a security policy, which enforces all inter-compartment access control checks specified by the SOP. Security decisions are made with respect to a compartment's security principal, which contains the origin and CSP of the compartment.

Since COWL's security model is very similar to this existing model, we can leverage these wrappers to introduce COWL's new security policies. We associate a label, clearance, and privilege with each compartment alongside the security principal. Wrappers consider all of these properties together when making security decisions.

Intra-Browser Confinement As shown in Table 1, we rely on wrappers to confine cross-compartment communication. Once confinement mode is enabled, we "recompute" all cross-compartment wrappers to use our MAC wrapper policy and thereby ensure that all subsequent cross-compartment access is mediated not only by the SOP, but also by confinement. For `postMessage`, our policy ensures that the receiver's label subsumes that of the sender (taking the receiver's privileges into consideration); otherwise the message is silently dropped. For a cross-compartment DOM property access, we additionally check that the sender's label subsumes that of the receiver—*i.e.,* that the labels of the compartments are equivalent after considering the sender's privileges (in addition to the same-origin check performed by the SOP).

Blink's execution contexts (the dual to Gecko's compartments) do not rely on wrappers to enforce cross-context access control. Instead, Blink implements the

[11] Since the Chromium architecture does not have cross-compartment wrappers, we modify the DOM binding code to insert label checks.

SOP security checks in the DOM binding code for a limited subset of DOM elements that may allow cross-origin access. Since COWL policies are more fine-grained, we modified the binding code to extend the security checks to all DOM objects and also perform label checks when confinement mode is enabled. Unfortunately, without wrappers, shared references cannot efficiently be revoked (*i.e.,* without walking the heap). Hence, before enabling confinement mode, a page can create a same-origin iframe with which it shares references, and the iframe can thereafter leak any data from the parent even if the latter's label is raised. To prevent this eventuality, our current Chromium API allows senders to disallow unlabeling Blobs if the target created any children before entering confinement mode.

Our implementations of LWorkers, whose API appears in Figure 6c, reuse labeled contexts straightforwardly. In fact, the `LWorker` constructor simply creates a new compartment with a fresh origin that contains a fresh JavaScript global object to which we attach the XHR constructor, COWL API, and primitives for communicating with the parent (*e.g.,* `postMessage`). Since LWorkers may have access to their parents' DOM, however, our wrappers distinguish them from other contexts to bypass SOP checks and only restrict DOM access according to MAC. This implementation is very similar to the content scripts used by Chrome and Firefox extensions [10, 26].

Browser-Server Confinement As shown in Table 1, we confine external communication (including XHR, content loading, and navigation) using CSP. While CSP alone is insufficient for providing flexible confinement,[12] it sufficiently addresses our external communication concern by precisely controlling from where a page loads content, performs XHR requests to, *etc.* To this end, we set a custom CSP policy whenever the compartment label changes, *e.g.,* with `COWL.label`. For instance, if the effective compartment label is `Label("https://bank.ch").and("https://amazon.com")`, all the underlying CSP directives are set to `'none'` (*e.g.,* `default-src 'none'`), disallowing all network communication. We also disable navigation with the `'sandbox'` directive [46–48].

Browser Storage Confinement As shown in Table 1, we use the `sandbox` directive to restrict access to storage (*e.g.,* cookies and HTML5 local storage [47]), as have other systems [5]. We leave the implementation of labeled storage as future work.

6 Evaluation

Performance largely determines acceptance of new browser features in practice. We evaluate the performance of COWL by measuring the cost of our new primitives as well as their impact on legacy web sites that do not use COWL's features. Our experiments consist of micro-benchmarks of API functions and end-to-end benchmarks of our example applications. We conducted all measurements on a 4-core i7-2620M machine with 16GB of RAM running GNU/Linux 3.13. The browser retrieved applications from the Node.js web server over the loopback interface. We note that these measurements are harsh for COWL, in that they omit network latency and the complex intra-context computation and DOM rendering of real-world applications, all of which would mask COWL's overhead further. Our key findings include:

- COWL's latency impact on legacy sites is negligible.
- Confining code with LWorkers is inexpensive, especially when compared to iframes/Workers. Indeed, the performance of our end-to-end confined password checker is only 5 ms slower than that of an inlined `script` version.
- COWL's incurs low overhead when enforcing confinement on mashups. The greatest overhead observed is 16% (for the encrypted document editor). Again, the absolute slowdown of 16 ms is imperceptible by users.

	Firefox			Chromium		
	vanilla	unlabeled	labeled	vanilla	unlabeled	labeled
New iframe	14.4	14.5	14.4	50.6	48.7	51.8
New worker	15.9	15.4	0.9†	18.9	18.9	3.3†
Iframe comm.	0.11	0.11	0.12	0.04	0.04	0.04
XHR comm	3.5	3.6	3.7	7.0	7.4	7.2
Worker comm.	0.20	0.24	0.03‡	0.07	0.07	0.03‡

Table 2: Micro-benchmarks, in milliseconds.

6.1 Micro-Benchmarks

Context Creation Table 2 shows micro-benchmarks for the stock browsers (vanilla), the COWL browsers with confinement mode turned off (unlabeled), and with confinement mode enabled (labeled). COWL adds negligible latency to compartment creation; indeed, except for LWorkers (†), the differences in creation times are of the order of measurement variability. We omit measurements of labeled "normal" Workers since they do not differ from those of unlabeled Workers. We attribute COWL's iframe-creation speedup in Chromium to measurement variability. We note that the cost of creating LWorkers is considerably less than that for "normal" Workers, which run in separate OS threads (†).

Communication The iframe, worker, and XHR communication measurements evaluate the round-trip latencies across iframes, workers, and the network. For the XHR benchmark, we report the cost of using the labeled XHR constructor averaged over 10,000 requests. Our

[12] There are two primary reasons. First, JavaScript code cannot (yet) modify a page's CSP. And, second, CSP does not (yet) provide a directive for restricting in-browser communication, *e.g.,* with `postMessage`.

Chromium implementation uses an LWorker to wrap the unmodified XHR constructor, so the cost of labeled XHR incorporates an additional cross-context call. As with creation, communicating with LWorkers (‡) is considerably faster than with "normal" Workers. This speedup arises because a lightweight LWorker shares an OS thread and event loop with their parent.

Labels We measured the cost of setting/getting the current label and the average cost of a label check in Firefox. For a randomly generated label with a handful of origins, these operations take on the order of one microsecond. The primary cost is recomputing cross-compartment wrappers and the underlying CSP policy, which ends up costing up to 13ms (*e.g.,* when the label is raised from public to a third-party origin). For many real applications, we expect raising the current label to be a rare occurrence. Moreover, there is much room for optimization (*e.g.,* porting COWL to the newest CSP implementation, which sets policies 15× faster [19]).

DOM We also executed the Dromaeo benchmark suite [29], which evaluates the performance of core functionality such as querying, traversing, and manipulating the DOM, in Firefox and Chromium. We found the performance of the vanilla and unlabeled browsers to be on par: the greatest slowdown was under 4%.

6.2 End-to-End Benchmarks

To focus on measuring COWL's overhead, we compare our apps against similarly compartmentalized but non-secure apps—*i.e.,* apps that perform no security checks.

Password-Strength Checker We measure the average duration of creating a new LWorker, fetching an 8 KB checker script based on [24], and checking a password sixteen characters in length. The checker takes an average of 18 ms (averaged over ten runs) on Firefox (labeled), 4 ms less than using a Worker on vanilla Firefox. Similarly, the checker running on labeled Chromium is 5 ms faster than the vanilla counterpart (measured at 54 ms). In both cases COWL achieves a speedup because its LWorkers are cheaper than normal Workers. However, these measurements are roughly 5 ms slower than simply loading the checker using an unsafe `script` tag.

Encrypted Document Editor We measure the end-to-end time taken to load the application and encrypt a 4 KB document using the SJCL AES-128 library [32]. The total run time includes the time taken to load the document editor page, which in turn loads the encryption-layer iframe, which further loads the editor proper. On Firefox (labeled) the workload completes in 116 ms; on vanilla Firefox, a simplified and unconfined version completes in 100ms. On Chromium, the performance measurements were comparable; the completion time was within 1ms of 244ms. The most expensive operation in the COWL-enabled Firefox app is raising the current label, since it requires changing the underlying document origin and recomputing the cross-compartment wrappers and CSP.

Third-Party Mashup We implemented a very simple third-party mashup application that makes a labeled XHR request to two unaffiliated origins, each of which produces a response containing a 27-byte JSON object with a numerical property, and sums the responses together. The corresponding vanilla app is identical, but uses the normal XHR object. In both cases we use CORS to permit cross-origin access. The Firefox (labeled) workload completes in 41 ms, which is 6 ms slower than the vanilla version. As in the document editor the slowdown derives from raising the current label, though in this case only for a single iframe. On Chromium (labeled) the workload completes in 55 ms, 2 ms slower than the vanilla one; the main slowdown here derives from our implementing labeled XHR with a wrapping LWorker.

Untrusted Third-Party Library We measured the load time of a banking application that incorporates jQuery and a library that traverses the DOM to replace phone numbers with links. The latter library uses XHR in attempt to leak the page's content. We compartmentalize the main page into a public outer component and a sensitive iframe containing the bank statement. In both compartments, we place the bank's trusted code (which loads the libraries) in a trusted labeled DOM worker with access to the page's DOM. We treat the rest of the code as untrusted. As our current Chromium implementation does not yet support DOM access for LWorkers, we only report measurements for Firefox. The measured latency on Firefox (labeled) is 165 ms, a 5 ms slowdown when compared to the unconfined version running on vanilla Firefox. Again, COWL prevents sensitive content from being exfiltrated and incurs negligible slowdown.

7 Discussion and Limitations

We now discuss the implications of certain facets of COWL's design, and limitations of the system.

User-Configured Confinement Recall that in the status-quo web security architecture, to allow cross-origin sharing, a server must grant individual foreign origins access to its data with CORS in an all-or-nothing, DAC fashion. COWL improves this state of affairs by allowing a COWL-aware server to more finely restrict how its shared data is disseminated—*i.e.,* when the server grants a foreign origin access to its data, it can confine the foreign origin's script(s) by setting a label on responses it sends the client.

Unfortunately, absent a permissive CORS header that whitelists the origins of applications that a user wishes to use, the SOP prohibits foreign origins from reading responses from the server, even in a COWL-enabled browser. Since a server's operator may not be aware of all applications its users may wish to use, the result is usually the same status-quo unpalatable choice between functionality and privacy—*e.g.,* give one's bank login credentials to Mint, or one cannot use the Mint application. For this reason, our COWL implementation lets browser users augment CORS by configuring for an origin (*e.g.,* `chase.com`) any foreign origins (*e.g.,* `mint.com`, `benjamins.biz`) they wish to additionally whitelist. In turn, COWL will confine these client-whitelisted origins (*e.g.,* `mint.com`) by labeling every response from the configured origin (`chase.com`). COWL obeys the server-supplied label when available and server whitelisting is *not* provided. Otherwise, COWL conservatively labels the response with a *fresh* origin (as described in Section 3.3). The latter ensures that once the response has been inspected, the code cannot communicate with *any* server, including at the *same* origin, since such requests carry the risks of self-exfiltration [11] and cross-site request forgery [39].

Covert Channels In an ideal confinement system, it would always be safe to let untrusted code compute on sensitive data. Unfortunately, real-world systems such as browsers typically exhibit *covert* channels that malicious code may exploit to exfiltrate sensitive data. Since COWL extends existing browsers, we do not protect against covert channel attacks. Indeed, malicious code can leverage covert channels already present in today's browsers to leak sensitive information. For instance, a malicious script within a confined context may be able to modulate sensitive data by varying rendering durations. A less confined context may then in turn exfiltrate the data to a remote host [20]. It is important to note, however, that COWL does not introduce new covert channels—our implementations re-purpose existing (software-based) browser isolation mechanisms (V8 contexts and SpiderMonkey compartments) to enforce MAC policies. Moreover, these MAC policies are generally more restricting than existing browser policies: they prevent unauthorized data exfiltration through *overt* channels and, in effect, force malicious code to resort to using covert channels.

The only fashion in which COWL relaxes status-quo browser policies is by allowing users to override CORS to permit cross-origin (labeled) sharing. Does this functionality introduce new risks? Whitelisting is user controlled (*e.g.,* the user must explicitly allow `mint.com` to read `amazon.com` and `chase.com` data), and code reading cross-origin data is subject to MAC (*e.g.,* `mint.com` cannot arbitrarily exfiltrate the `amazon.com` or `chase.com` data after reading it). In contrast, today's mashups like `mint.com` ask users for their passwords. COWL is strictly an improvement: under COWL, when a user decides to trust a mashup integrator such as `mint.com`, she *only* trusts the app to not leak her data through covert channels. Nevertheless, users can make poor security choices. Whitelisting malicious origins would be no exception; we recognize this as a limitation of COWL that must be communicated to the end-user.

A trustworthy developer can leverage COWL's support for *clearance* when compartmentalizing his application to ensure that only code that actually relies on cross-origin data has access to it. Clearance is a label that serves as an upper bound on a context's current label. Since COWL ensures that the current label is adjusted according to the sensitivity of the data being read, code cannot read (and thus leak) data labeled above the clearance. Thus, Mint can assign a "low" clearance to untrusted third-party libraries, *e.g.,* to keep `chase.com`'s data confidential. These libraries will then not be able to leak such data through covert channels, even if they are malicious.

Expressivity of Label Model COWL uses DC labels [33] to enforce confinement according to an information flow control discipline. Although this approach captures a wide set of confinement policies, it is not expressive enough to handle policies with a circular flow of information [6] or some policies expressible in more powerful logics (*e.g.,* first order logic, as used by Nexus [30]). DC labels are, however, as expressive as other popular label models [25], including Myers and Liskov's Decentralized Label Model [27]. Our experience implementing security policies with them thus far suggests they are expressive enough to support featureful web applications.

We adopted DC labels largely because their fit with web origins pays practical dividends. First, as developers already typically express policies by whitelisting origins, we believe they will find DC labels intuitive to use. Second, because both DC labels and today's web policies are defined in terms of origins, the implementation of COWL can straightforwardly reuse the implementation of existing security mechanisms, such as CSP.

8 Related Work

Existing browser confinement systems based on information flow control can be classified either as *fine-grained* or *coarse-grained*. The former associate IFC policies with individual objects, while the latter associate policies with entire browsing contexts. We compare COWL to previously proposed systems in both categories, then contrast the two categories' overall characteristics.

Coarse-grained IFC COWL shares many features with existing coarse-grained systems. BFlow [50], for example, allows web sites to enforce confinement policies stricter than the SOP via *protection zones*—groups of iframes sharing a common label. However, BFlow cannot mediate between mutually distrustful principals—*e.g.,* the encrypted document editor is not directly implementable with BFlow. This is because only asymmetric confinement is supported—a sub-frame cannot impose any restrictions on its parent. For the same reasons, BFlow cannot support applications that require security policies more flexible than the SOP, such as our third-party mashup example. These differences reflect different goals for the two systems. BFlow's authors set out to confine untrusted third-party scripts, while we also seek to support applications that incorporate code from mutually distrusting parties.

More recently, Akhawe *et al.* propose the data-confined sandbox (DCS) system [5], which allows pages to intercept and monitor the network, storage, and cross-origin channels of `data:` URI iframes. The limitation to `data:` URI iframes means DCS cannot confine the common case of a service provided in an iframe [31]. Like BFlow, DCS does not offer symmetric confinement, and does not incorporate functionality to let developers build applications like third-party mashups.

Fine-grained IFC Per-object-granularity IFC makes it easier to confine untrusted libraries that are closely coupled with trusted code on a page (*e.g.,* jQuery) and avoid the problem of *over-tainting,* where a single context accumulates taint as it inspects more data.

JSFlow [15] is one such fine-grained JavaScript IFC system, which enforces policies by executing JavaScript in an interpreter written in JavaScript. This approach incurs a two order of magnitude slowdown. JSFlow's authors suggest that this cost makes JSFlow a better fit for use as a development tool than as an "always-on" privacy system for users' browsers. Additionally, JSFlow does not support applications that rely on policies more flexible than the SOP, such as our third-party mashup example.

The FlowFox fine-grained IFC system [12] enforces policies with secure-multi execution (SME) [13]. SME ensures that no leaks from a sensitive context can leak into a less sensitive context by executing a program multiple times. Unlike JSFlow and COWL, SME is not amenable to scenarios where declassification plays a key role (*e.g.,* the encrypted editor or the password manager). FlowFox's labeling of user interactions and metadata (history, screen size, *etc.*) do allow it to mitigate history sniffing and behavior tracking; COWL does not address these attacks.

While fine-grained IFC systems may be more convenient for developers, they impose new language semantics for developers to learn, require invasive modifications to the JavaScript engine, and incur greater performance overhead. In contrast, because COWL repurposes familiar isolation constructs and does not require JavaScript engine modifications, it is relatively straightforward to add to legacy browsers. It also only adds overhead to cross-compartment operations, rather than to all JavaScript execution. The typically short lifetime of a browsing context helps avoid excessive accumulation of taint. We conjecture that coarse-grained and fine-grained IFC are equally expressive, provided one may use arbitrarily many compartments—a cost in programmer convenience. Finally, coarse- and fine-grained mechanisms are not mutually exclusive. For instance, to confine legacy (non-compartmentalized) JavaScript code, one could deploy JSFlow within a COWL context.

Sandboxing The literature on sandboxing and secure subsets of JavaScript is rich, and includes Caja [1], BrowserShield [28], WebJail [37], TreeHouse [18], JSand [4], SafeScript [36], Defensive JavaScript [9], and Embassies [16]). While our design has been inspired by some of these systems (*e.g.,* TreeHouse), the usual goals of these systems are to mediate security-critical operations, restrict access to the DOM, and restrict communication APIs. In contrast to the mandatory nature of confinement, however, these systems impose most restrictions in discretionary fashion, and are thus not suitable for building some of the applications we consider (in particular, the encrypted editor). Nevertheless, we believe that access control and language subsets are crucial complements to confinement for building robustly secure applications.

9 Conclusion

Web applications routinely pull together JavaScript contributed by parties untrusted by the user, as well as by mutually distrusting parties. The lack of confinement for untrusted code in the status-quo browser security architecture puts users' privacy at risk. In this paper, we have presented COWL, a label-based MAC system for web browsers that preserves users' privacy in the common case where untrusted code computes over sensitive data. COWL affords developers flexibility in synthesizing web applications out of untrusted code and services while preserving users' privacy. Our positive experience building four web applications atop COWL for which privacy had previously been unattainable in status-quo web browsers suggests that COWL holds promise as a practical platform for preserving privacy in today's pastiche-like web applications. And our measurements of COWL's performance overhead in the Firefox and Chromium browsers suggest that COWL's privacy benefits come at negligible end-to-end cost in performance.

Acknowledgements

We thank Bobby Holley, Blake Kaplan, Ian Melven, Garret Robinson, Brian Smith, and Boris Zbarsky for helpful discussions of the design and implementation of COWL. We thank Stefan Heule and John Mitchell for useful comments on formal aspects of the design. And we thank our shepherd Mihai Budiu and the anonymous reviewers for their helpful comments. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, by the EPSRC under grant EP/K032542/1, the Swedish research agencies VR and STINT, the Barbro Osher Pro Suecia foundation, and by multiple gifts from Google (to Stanford and UCL). Deian Stefan and Edward Z. Yang are supported through the NDSEG Fellowship Program.

References

[1] Google Caja. A source-to-source translator for securing JavaScript-based web content. http://code.google.com/p/google-caja/, 2013.

[2] Mint. http://www.mint.com/, 2013.

[3] jQuery Usage Statistics: Websites using jQuery. http://trends.builtwith.com/javascript/jQuery, 2014.

[4] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *ACSAC*, 2012.

[5] D. Akhawe, F. Li, W. He, P. Saxena, and D. Song. Data-confined HTML5 applications. In *ESORICS*, 2013.

[6] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical domain and type enforcement for UNIX. In *Security and Privacy*, 1995.

[7] A. Barth. The web origin concept. Technical report, IETF, 2011. URL https://tools.ietf.org/html/rfc6454.

[8] A. Barth, C. Jackson, and J. Mitchell. Securing frame communication in browsers. *Communications of the ACM*, 52(6):83–91, 2009.

[9] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Language-based defenses against untrusted browser origins. In *USENIX Security*, 2013.

[10] N. Carlini, A. P. Felt, and D. Wagner. An evaluation of the Google Chrome extension security architecture. In *USENIX Security*, 2012.

[11] E. Y. Chen, S. Gorbaty, A. Singhal, and C. Jackson. Self-exfiltration: The dangers of browser-enforced information flow control. In *Web 2.0 Security and Privacy*, 2012.

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

[13] D. Devriese and F. Piessens. Noninterference through Secure Multi-Execution. In *Security and Privacy*, 2010.

[14] 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 *OSDI*, 2005.

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

[16] J. Howell, B. Parno, and J. R. Douceur. Embassies: Radically refactoring the Web. In *NSDI*, 2013.

[17] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your ifcexception are belong to us. In *Security and Privacy*, 2013.

[18] L. Ingram and M. Walfish. Treehouse: JavaScript sandboxes to help web developers help themselves. In *USENIX ATC*, 2012.

[19] C. Kerschbaumer. Faster Content Security Policy (CSP). https://blog.mozilla.org/security/2014/09/10/faster-csp/, 2014.

[20] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson. Cross-origin pixel stealing: timing attacks using CSS filters. In *CCS*, 2013.

[21] M. S. Miller. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, 2006.

[22] M. S. Miller and J. S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *ASIAN*, 2003.

[23] M. S. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://zesty.ca/capmyths/usenix.pdf.

[24] S. Moitozo. http://www.geekwisdom.com/js/passwordmeter.js, 2006.

[25] B. Montagu, B. C. Pierce, and R. Pollack. A theory of information-flow labels. In *CSF*, June 2013.

[26] Mozilla. Add-on builder and SDK. https://addons.mozilla.org/en-US/developers/docs/sdk/, 2013.

[27] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *TOSEM*, 9(4), 2000.

[28] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic HTML. *TWEB*, 1(3), Sept. 2007.

[29] J. Reisg. Dromaeo: JavaScript performance testing. http://dromaeo.com/, 2014.

[30] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *SOSP*, 2011.

[31] S. Son and V. Shmatikov. The postman always rings twice: Attacking and defending postMessage in HTML5 websites. In *NDSS*, 2013.

[32] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in JavaScript. In *ACSAC*, 2009.

[33] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *NordSec*, 2011.

[34] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*, 2011.

[35] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *ICFP*, 2012.

[36] M. Ter Louw, P. H. Phung, R. Krishnamurti, and V. N. Venkatakrishnan. SafeScript: JavaScript transformation for policy enforcement. In *Secure IT Systems*, 2013.

[37] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail: least-privilege integration of third-party components in web mashups. In *ACSAC*, 2011.

[38] A. Van Kesteren. Cross-Origin Resource Sharing. `http://www.w3.org/TR/cors/`, 2012.

[39] B. Vibber. CSRF token-stealing attack (user.tokens). `https://bugzilla.wikimedia.org/show_bug.cgi?id=34907`, 2014.

[40] G. Wagner, A. Gal, C. Wimmer, B. Eich, and M. Franz. Compartmental memory management in a modern web browser. *SIGPLAN Notices*, 46 (11), 2011.

[41] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. *ACM SIGOPS Operating Systems Review*, 41(6), 2007.

[42] WC3. Content Security Policy 1.0. `http://www.w3.org/TR/CSP/`, 2012.

[43] WC3. HTML5 web messaging. `http://www.w3.org/TR/webmessaging/`, 2012.

[44] WC3. Web Workers. `http://www.w3.org/TR/workers/`, 2012.

[45] WC3. Cross-Origin Resource Sharing. `http://www.w3.org/TR/cors/`, 2013.

[46] WC3. Content Security Policy 1.1. `https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`, 2013.

[47] WC3. HTML5. `http://www.w3.org/TR/html5/`, 2013.

[48] WHATWG. HTML living standard. `http://developers.whatwg.org/`, 2013.

[49] E. Yang, D. Stefan, J. Mitchell, D. Mazières, P. Marchenko, and B. Karp. Toward principled browser security. In *HotOS*, 2013.

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

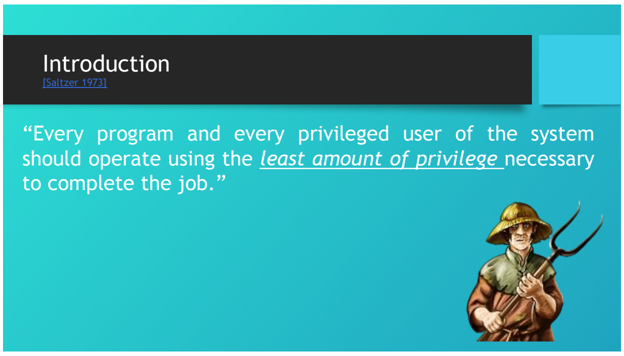
[51] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

[52] M. Zelwski. Browser security handbook, part 2. `HTtp://code.google.com/p/browsersec/wiki/Part2`, 2011.

Web Server Security

The Principle of Least Privilege

Alejandro Russo
russo@chalmers.se
ECI 2015, UBA, Buenos Aires, Argentina



Introduction

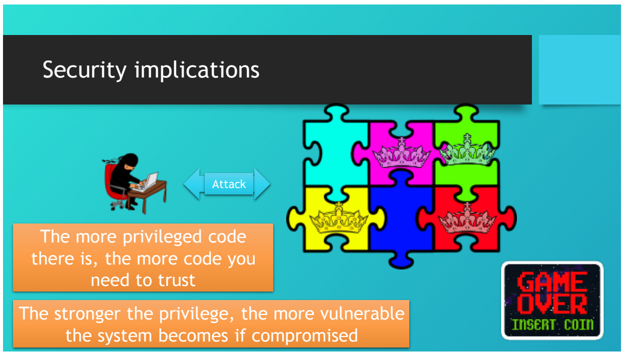
[Saltzer 1973]

"Every program and every privileged user of the system should operate using the *least amount of privilege* necessary to complete the job."

What happens in practice?

"The principle of most privilege"

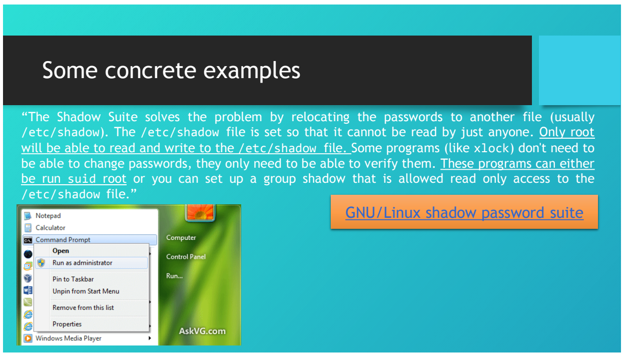
MAKE ME A SANDWICH.
WHAT? MAKE IT YOURSELF.
SUDO MAKE ME A SANDWICH.
OKAY.



Security implications

The more privileged code there is, the more code you need to trust

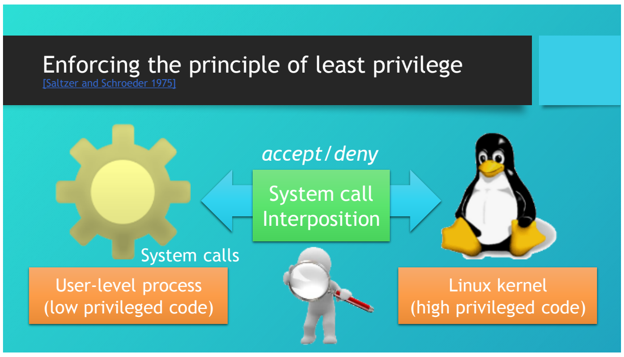
The stronger the privilege, the more vulnerable the system becomes if compromised



Some concrete examples

"The Shadow Suite solves the problem by relocating the passwords to another file (usually `/etc/shadow`). The `/etc/shadow` file is set so that it cannot be read by just anyone. Only root will be able to read and write to the `/etc/shadow` file. Some programs (like `xlock`) don't need to be able to change passwords, they only need to be able to verify them. These programs can either be run `suid root` or you can set up a group shadow that is allowed read only access to the `/etc/shadow` file."

GNU/Linux shadow password suite



Enforcing the principle of least privilege

[Saltzer and Schroeder 1975]

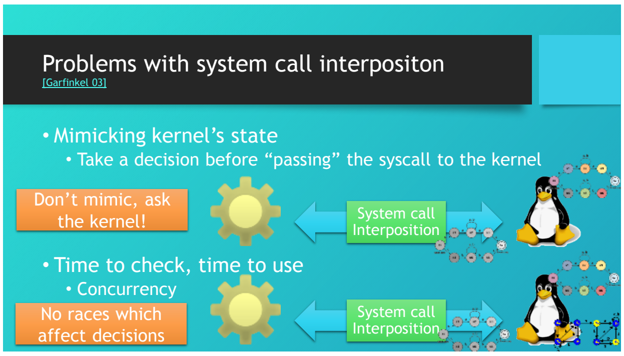
accept/deny

System call Interposition

System calls

User-level process (low privileged code)

Linux kernel (high privileged code)



Problems with system call interpositon

[Garfinkel 03]

- Mimicking kernel's state
  - Take a decision before "passing" the syscall to the kernel

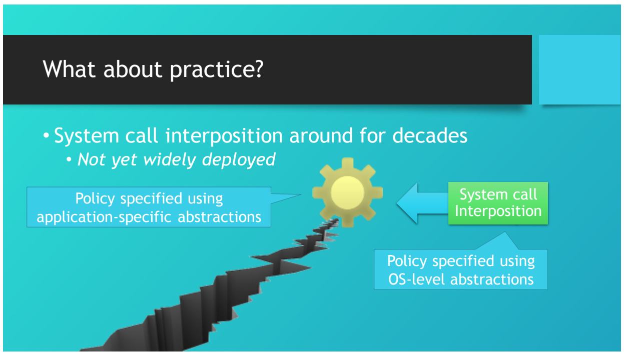
Don't mimic, ask the kernel!

System call Interposition

- Time to check, time to use
  - Concurrency

No races which affect decisions

System call Interposition



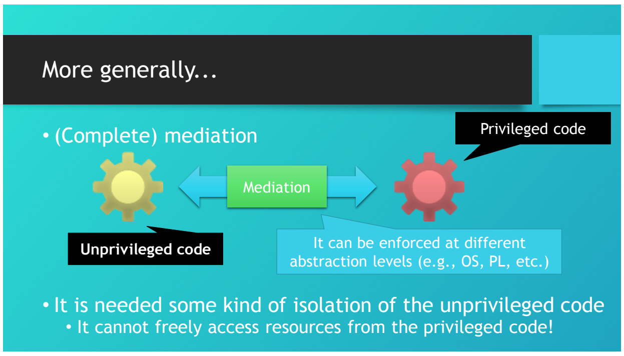
What about practice?

- System call interposition around for decades
  - *Not yet widely deployed*

Policy specified using application-specific abstractions

System call Interposition

Policy specified using OS-level abstractions



More generally...

- (Complete) mediation

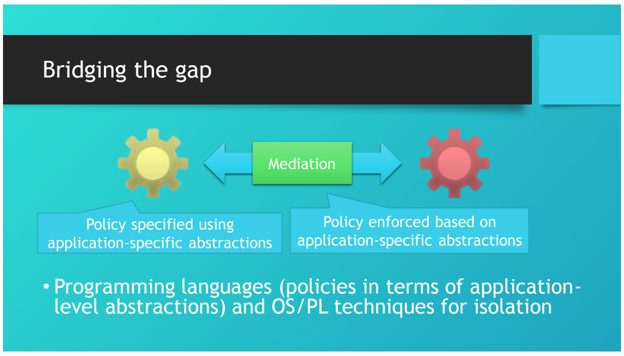
Privileged code

Mediation

Unprivileged code

It can be enforced at different abstraction levels (e.g., OS, PL, etc.)

- It is needed some kind of isolation of the unprivileged code
  - It cannot freely access resources from the privileged code!



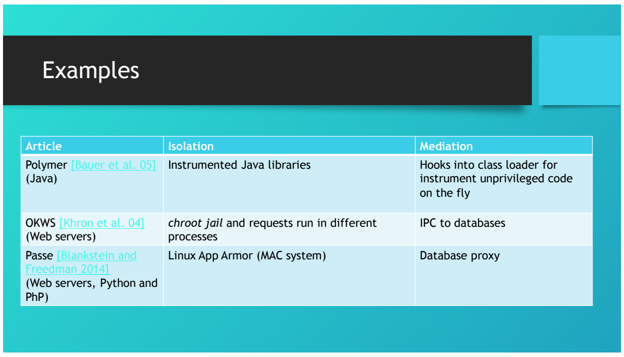
Bridging the gap

Mediation

Policy specified using application-specific abstractions

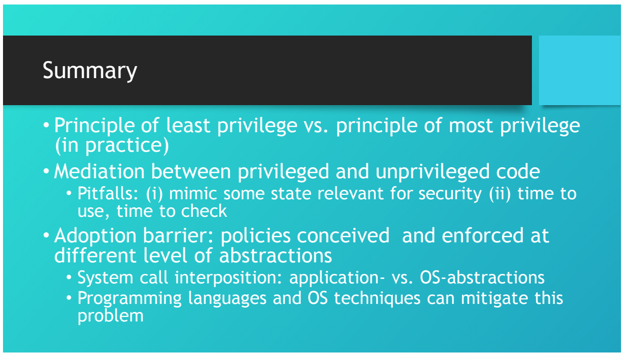
Policy enforced based on application-specific abstractions

- Programming languages (policies in terms of application-level abstractions) and OS/PL techniques for isolation



Examples

Article	Isolation	Mediation
Polymer [Bauer et al. 05] (Java)	Instrumented Java libraries	Hooks into class loader for instrument unprivileged code on the fly
OKWS [Khron et al. 04] (Web servers)	*chroot jail* and requests run in different processes	IPC to databases
Passe [Blankstein and Freedman 2014] (Web servers, Python and PhP)	Linux App Armor (MAC system)	Database proxy

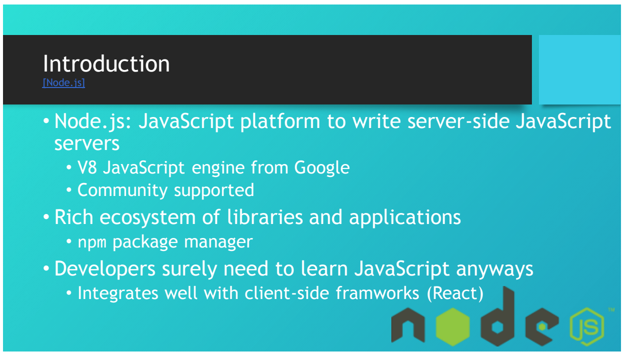


Summary

- Principle of least privilege vs. principle of most privilege (in practice)
- Mediation between privileged and unprivileged code
  - Pitfalls: (i) mimic some state relevant for security (ii) time to use, time to check
- Adoption barrier: policies conceived and enforced at different level of abstractions
  - System call interposition: application- vs. OS-abstractions
  - Programming languages and OS techniques can mitigate this problem

ESpectro (Node.js)

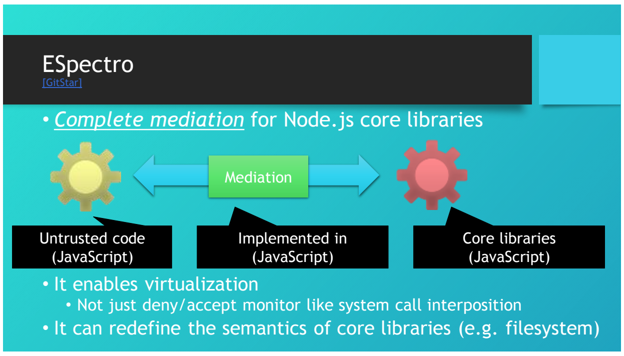
Alejandro Russo
russo@chalmers.se
ECI 2015, UBA, Buenos Aires, Argentina



Introduction

[Node.js]

- Node.js: JavaScript platform to write server-side JavaScript servers
  - V8 JavaScript engine from Google
  - Community supported
- Rich ecosystem of libraries and applications
  - npm package manager
- Developers surely need to learn JavaScript anyways
  - Integrates well with client-side framworks (React)



ESpectro

[GitStar]

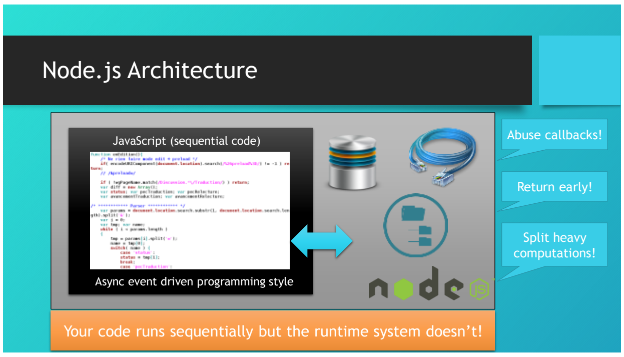
- *Complete mediation* for Node.js core libraries

Mediation
Untrusted code (JavaScript)
Implemented in (JavaScript)
Core libraries (JavaScript)

- It enables virtualization
  - Not just deny/accept monitor like system call interposition
- It can redefine the semantics of core libraries (e.g. filesystem)

Authors

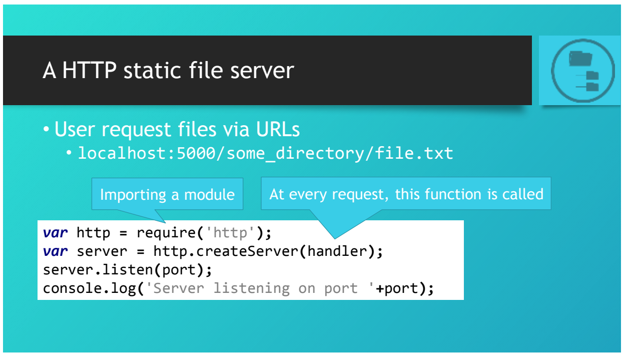
- Startup in San Francisco (4 people)
- CTO: Deian Stefan (LIO, Hails, COWL)
- ESpectro is work in progress
  - Exclusive preview in this course!



Node.js Architecture

JavaScript (sequential code)
Async event driven programming style
Abuse callbacks!
Return early!
Split heavy computations!

Your code runs sequentially but the runtime system doesn't!

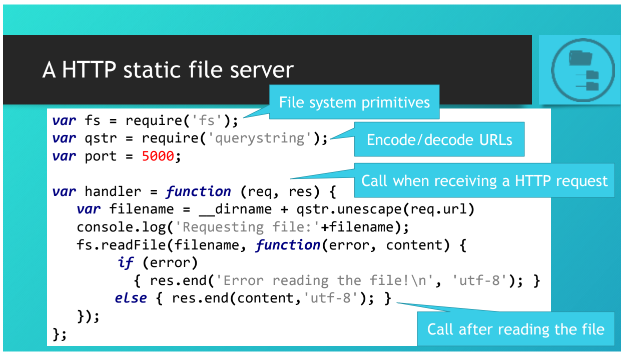


A HTTP static file server

- User request files via URLs
  - localhost:5000/some_directory/file.txt

Importing a module
At every request, this function is called

```
var http = require('http');
var server = http.createServer(handler);
server.listen(port);
console.log('Server listening on port '+port);
```



A HTTP static file server

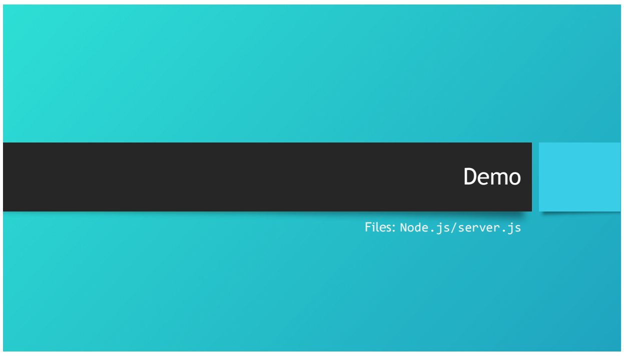
```
var fs = require('fs');
var qstr = require('querystring');
var port = 5000;

var handler = function (req, res) {
   var filename = __dirname + qstr.unescape(req.url)
   console.log('Requesting file:'+filename);
   fs.readFile(filename, function(error, content) {
        if (error)
          { res.end('Error reading the file!\n', 'utf-8'); }
        else { res.end(content,'utf-8'); }
   });
};
```

File system primitives

Encode/decode URLs

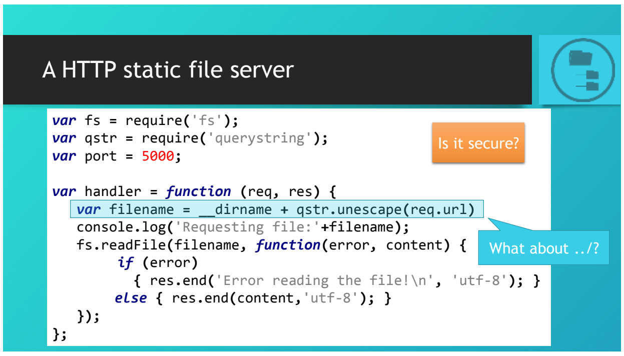
Call when receiving a HTTP request

Call after reading the file



Demo

Files: Node.js/server.js

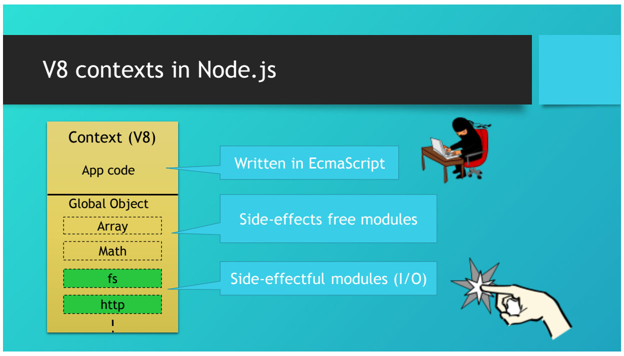


A HTTP static file server

```
var fs = require('fs');
var qstr = require('querystring');
var port = 5000;

var handler = function (req, res) {
   var filename = __dirname + qstr.unescape(req.url)
   console.log('Requesting file:'+filename);
   fs.readFile(filename, function(error, content) {
        if (error)
          { res.end('Error reading the file!\n', 'utf-8'); }
        else { res.end(content,'utf-8'); }
   });
};
```

Is it secure?

What about ../?



V8 contexts in Node.js

Context (V8)

App code

Global Object

Array

Math

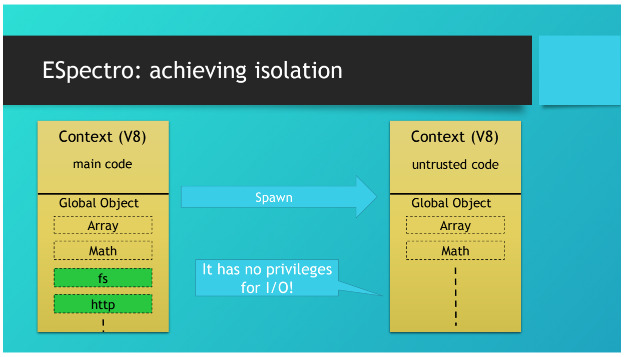
fs

http

Written in EcmaScript

Side-effects free modules

Side-effectful modules (I/O)



ESpectro: achieving isolation

Context (V8)

main code

Global Object

Array

Math

fs

http

Spawn

Context (V8)

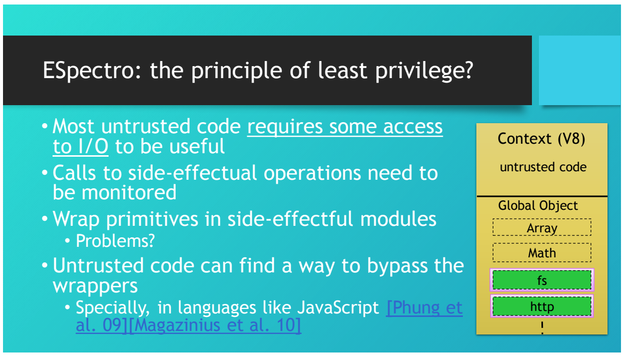
untrusted code

Global Object

Array

Math

It has no privileges for I/O!



ESpectro: the principle of least privilege?

- Most untrusted code requires some access to I/O to be useful
- Calls to side-effectual operations need to be monitored
- Wrap primitives in side-effectful modules
  - Problems?
- Untrusted code can find a way to bypass the wrappers
  - Specially, in languages like JavaScript [Phung et al. 09][Magazinius et al. 10]

Context (V8)

untrusted code

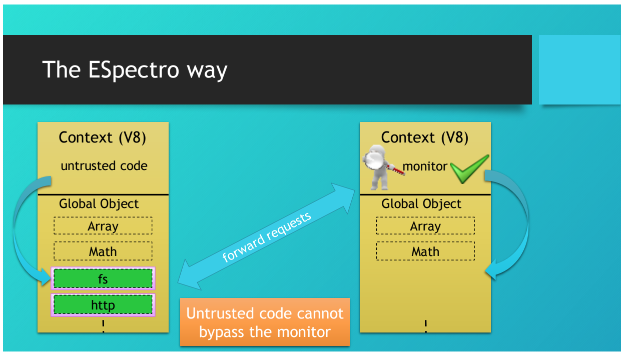
Global Object

Array

Math

fs

http



The ESpectro way

Context (V8)

untrusted code

Global Object

Array

Math

fs

http

forward requests

Untrusted code cannot bypass the monitor

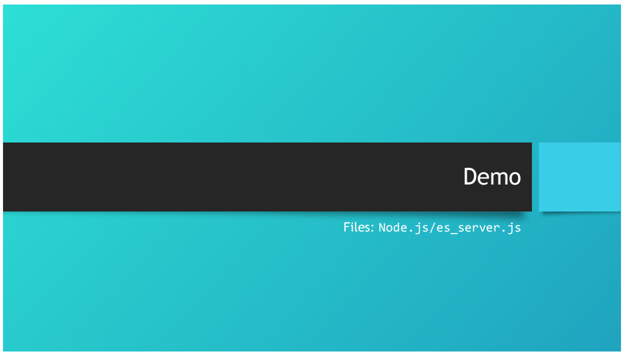
Context (V8)

monitor

Global Object

Array

Math



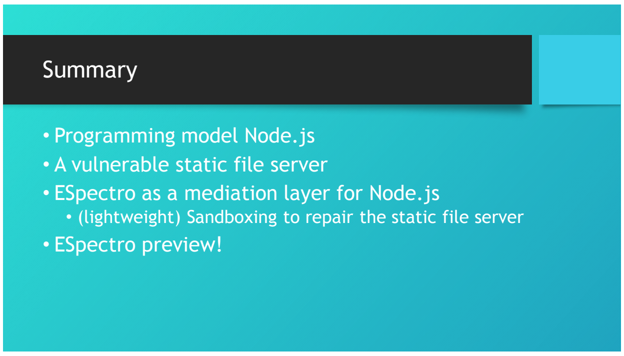
Demo

Files: `Node.js/es_server.js`

Mediation as a discipline technique

- Mediation has a tremendous expressive power
  - (Some form of) Sandboxing
  - Transparent encryption
  - Automatic sanitization
  - Mandatory Access Control
- If interested to try ESpectro beyond this course, contact GitStar!

GitStar



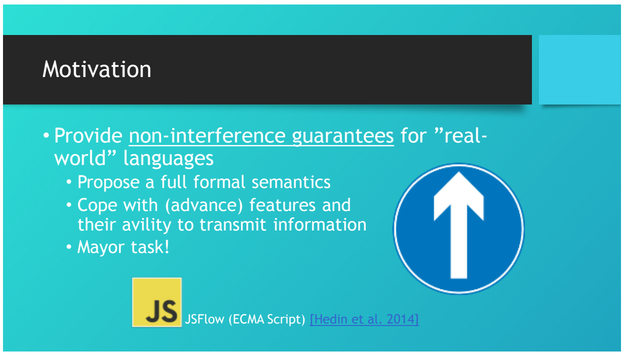
Summary

- Programming model Node.js
- A vulnerable static file server
- ESpectro as a mediation layer for Node.js
  - (lightweight) Sandboxing to repair the static file server
- ESpectro preview!

Formal Aspects

IFC-Inside

Alejandro Russo
russo@chalmers.se
ECI 2015, UBA, Buenos Aires, Argentina



Motivation

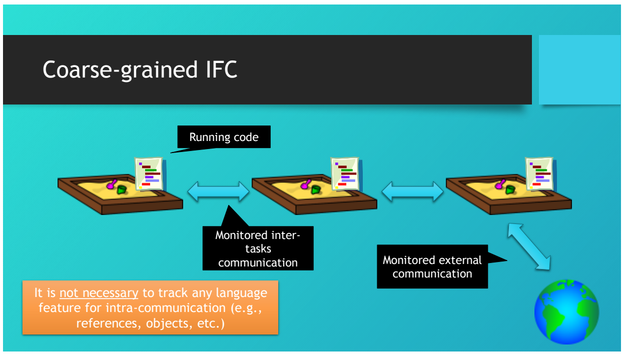
- Provide non-interference guarantees for ”real-world” languages
  - Propose a full formal semantics
  - Cope with (advance) features and their avility to transmit information
  - Mayor task!

JSFlow (ECMA Script) [Hedin et al. 2014]

IFC-Inside

[Heule et al. 2015]

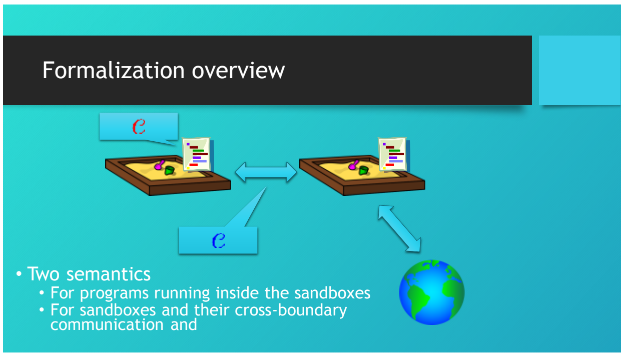
- We take an alternative road
- There exists many programming languages which support sandboxes
  - HTML (iFrames)
  - JavaScript (workers)
  - C / C++
- Sanboxes provide isolation from the host
- We can leverage on sandboxes to provide IFC!



Coarse-grained IFC

Running code
Monitored inter-tasks communication
Monitored external communication

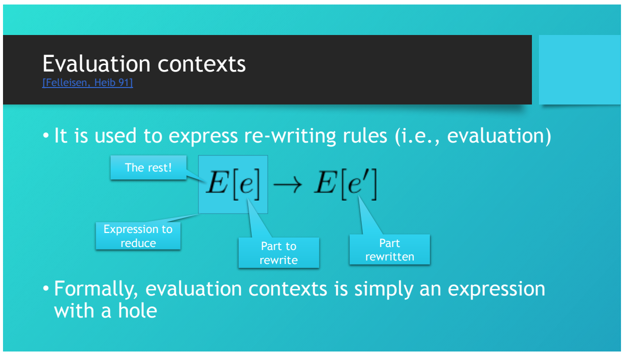
It is not necessary to track any language feature for intra-communication (e.g., references, objects, etc.)



Formalization overview

- Two semantics
  - For programs running inside the sandboxes
  - For sandboxes and their cross-boundary communication and

Semantics



Evaluation contexts

[Felleisen, Helb 91]

- It is used to express re-writing rules (i.e., evaluation)

$$E[e] \rightarrow E[e']$$

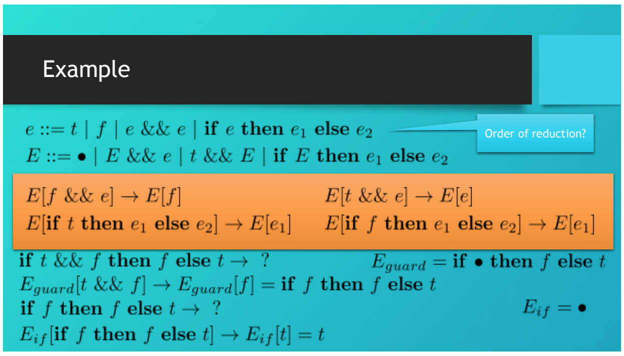
The rest!

Expression to reduce

Part to rewrite

Part rewritten

- Formally, evaluation contexts is simply an expression with a hole



Example

$e ::= t \mid f \mid e \;\&\&\; e \mid \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2$

$E ::= \bullet \mid E \;\&\&\; e \mid t \;\&\&\; E \mid \textbf{if}\ E\ \textbf{then}\ e_1\ \textbf{else}\ e_2$

Order of reduction?

$E[f \;\&\&\; e] \rightarrow E[f]$ $\qquad E[t \;\&\&\; e] \rightarrow E[e]$

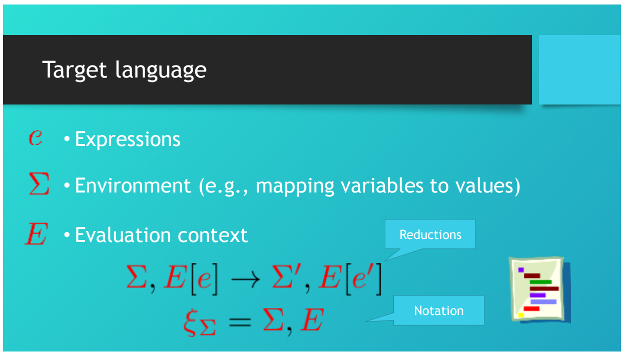
$E[\textbf{if}\ t\ \textbf{then}\ e_1\ \textbf{else}\ e_2] \rightarrow E[e_1]$ $\qquad E[\textbf{if}\ f\ \textbf{then}\ e_1\ \textbf{else}\ e_2] \rightarrow E[e_1]$

$\textbf{if}\ t \;\&\&\; f\ \textbf{then}\ f\ \textbf{else}\ t \rightarrow\ ?$ $\qquad E_{guard} = \textbf{if}\ \bullet\ \textbf{then}\ f\ \textbf{else}\ t$

$E_{guard}[t \;\&\&\; f] \rightarrow E_{guard}[f] = \textbf{if}\ f\ \textbf{then}\ f\ \textbf{else}\ t$

$\textbf{if}\ f\ \textbf{then}\ f\ \textbf{else}\ t \rightarrow\ ?$ $\qquad E_{if} = \bullet$

$E_{if}[\textbf{if}\ f\ \textbf{then}\ f\ \textbf{else}\ t] \rightarrow E_{if}[t] = t$



Target language

e • Expressions

Σ • Environment (e.g., mapping variables to values)

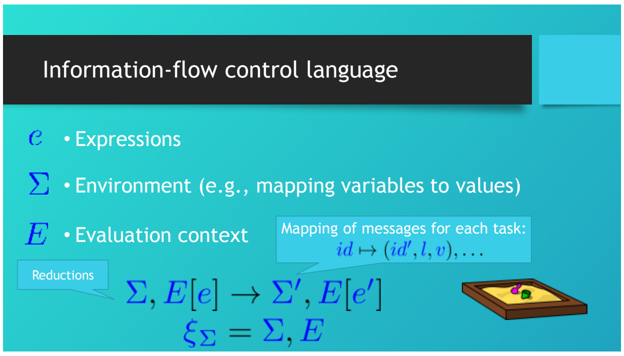
E • Evaluation context

Reductions

$$\Sigma, E[e] \rightarrow \Sigma', E[e']$$

$$\xi_\Sigma = \Sigma, E$$

Notation



Information-flow control language

e • Expressions

Σ • Environment (e.g., mapping variables to values)

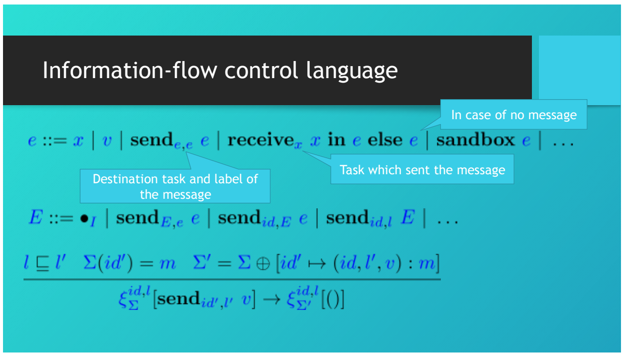
E • Evaluation context

Mapping of messages for each task: $id \mapsto (id', l, v), \ldots$

Reductions

$$\Sigma, E[e] \rightarrow \Sigma', E[e']$$

$$\xi_\Sigma = \Sigma, E$$



Information-flow control language

In case of no message

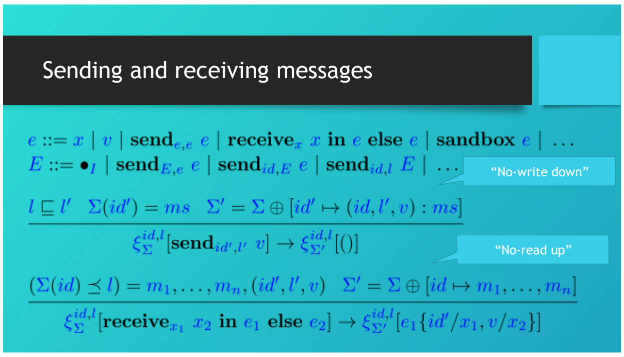
$e ::= x \mid v \mid \textbf{send}_{e,e}\ e \mid \textbf{receive}_x\ x\ \textbf{in}\ e\ \textbf{else}\ e \mid \textbf{sandbox}\ e \mid \ \ldots$

Destination task and label of the message

Task which sent the message

$E ::= \bullet_I \mid \textbf{send}_{E,e}\ e \mid \textbf{send}_{id,E}\ e \mid \textbf{send}_{id,l}\ E \mid \ \ldots$

$$\frac{l \sqsubseteq l' \quad \Sigma(id') = m \quad \Sigma' = \Sigma \oplus [id' \mapsto (id, l', v) : m]}{\xi_\Sigma^{id,l}[\textbf{send}_{id',l'}\ v] \rightarrow \xi_{\Sigma'}^{id,l}[()]}$$



Sending and receiving messages

$e ::= x \mid v \mid \textbf{send}_{e,e}\ e \mid \textbf{receive}_x\ x\ \textbf{in}\ e\ \textbf{else}\ e \mid \textbf{sandbox}\ e \mid \ \ldots$

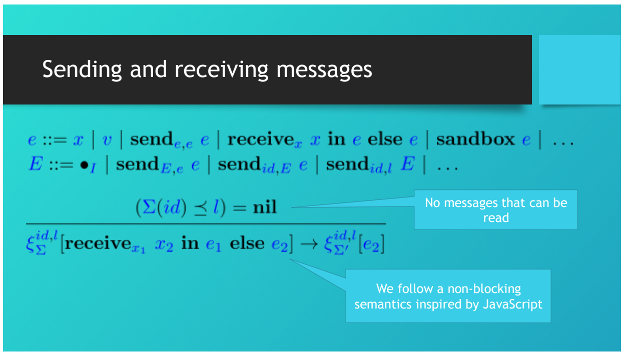
$E ::= \bullet_I \mid \textbf{send}_{E,e}\ e \mid \textbf{send}_{id,E}\ e \mid \textbf{send}_{id,l}\ E \mid \ \ldots$

“No-write down”

$$\frac{l \sqsubseteq l' \quad \Sigma(id') = ms \quad \Sigma' = \Sigma \oplus [id' \mapsto (id, l', v) : ms]}{\xi_\Sigma^{id,l}[\textbf{send}_{id',l'}\ v] \rightarrow \xi_{\Sigma'}^{id,l}[()]}$$

“No-read up”

$$\frac{(\Sigma(id) \preceq l) = m_1, \ldots, m_n, (id', l', v) \quad \Sigma' = \Sigma \oplus [id \mapsto m_1, \ldots, m_n]}{\xi_\Sigma^{id,l}[\textbf{receive}_{x_1}\ x_2\ \textbf{in}\ e_1\ \textbf{else}\ e_2] \rightarrow \xi_{\Sigma'}^{id,l}[e_1\{id'/x_1, v/x_2\}]}$$



Sending and receiving messages

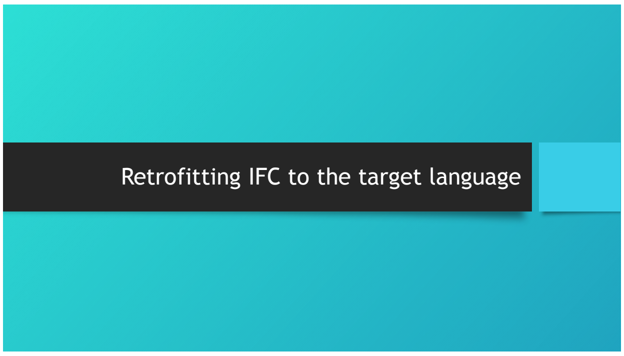
$e ::= x \mid v \mid \mathbf{send}_{e,e}\ e \mid \mathbf{receive}_x\ x\ \mathbf{in}\ e\ \mathbf{else}\ e \mid \mathbf{sandbox}\ e \mid \ \dots$

$E ::= \bullet_I \mid \mathbf{send}_{E,e}\ e \mid \mathbf{send}_{id,E}\ e \mid \mathbf{send}_{id,l}\ E \mid \ \dots$

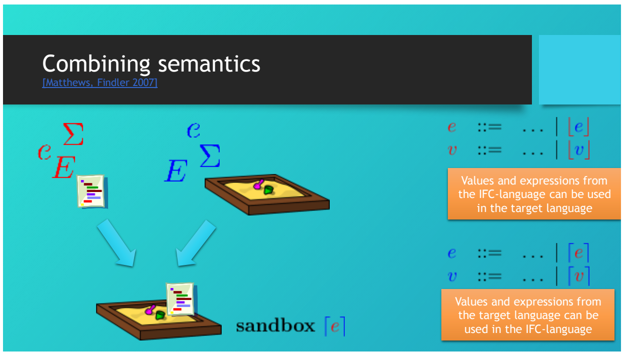
$$\frac{(\Sigma(id) \preceq l) = \mathbf{nil}}{\xi^{id,l}_{\Sigma}[\mathbf{receive}_{x_1}\ x_2\ \mathbf{in}\ e_1\ \mathbf{else}\ e_2] \to \xi^{id,l}_{\Sigma'}[e_2]}$$

No messages that can be read

We follow a non-blocking semantics inspired by JavaScript



Retrofitting IFC to the target language



Combining semantics

[Matthews, Findler 2007]

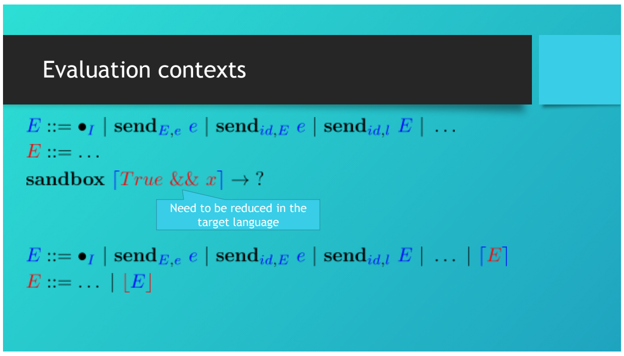
$\mathbf{sandbox}\ \lceil e \rceil$

$e \ ::= \ \dots \mid \lfloor e \rfloor$
$v \ ::= \ \dots \mid \lfloor v \rfloor$

Values and expressions from the IFC-language can be used in the target language

$e \ ::= \ \dots \mid \lceil e \rceil$
$v \ ::= \ \dots \mid \lceil v \rceil$

Values and expressions from the target language can be used in the IFC-language



Evaluation contexts

$E ::= \bullet_I \mid \mathbf{send}_{E,e}\ e \mid \mathbf{send}_{id,E}\ e \mid \mathbf{send}_{id,l}\ E \mid \ \dots$

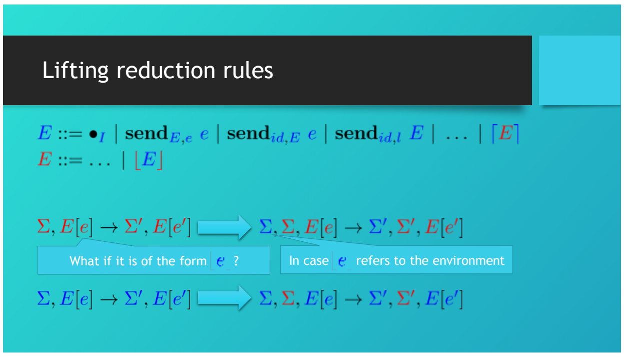
$E ::= \dots$

$\mathbf{sandbox}\ \lceil True\ \&\&\ x \rceil \to ?$

Need to be reduced in the target language

$E ::= \bullet_I \mid \mathbf{send}_{E,e}\ e \mid \mathbf{send}_{id,E}\ e \mid \mathbf{send}_{id,l}\ E \mid \ \dots \mid \lceil E \rceil$

$E ::= \dots \mid \lfloor E \rfloor$



Lifting reduction rules

$E ::= \bullet_I \mid \mathbf{send}_{E,e}\ e \mid \mathbf{send}_{id,E}\ e \mid \mathbf{send}_{id,l}\ E \mid \ \dots \mid \lceil E \rceil$

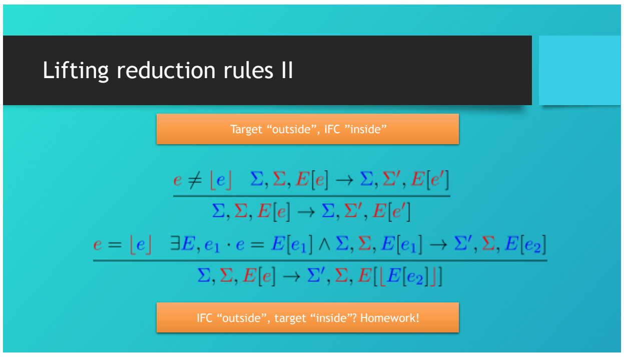
$E ::= \dots \mid \lfloor E \rfloor$

$\Sigma, E[e] \to \Sigma', E[e'] \Longrightarrow \Sigma, \Sigma, E[e] \to \Sigma', \Sigma', E[e']$

What if it is of the form $\lfloor e \rfloor$?

In case $\lfloor e \rfloor$ refers to the environment

$\Sigma, E[e] \to \Sigma', E[e'] \Longrightarrow \Sigma, \Sigma, E[e] \to \Sigma', \Sigma', E[e']$



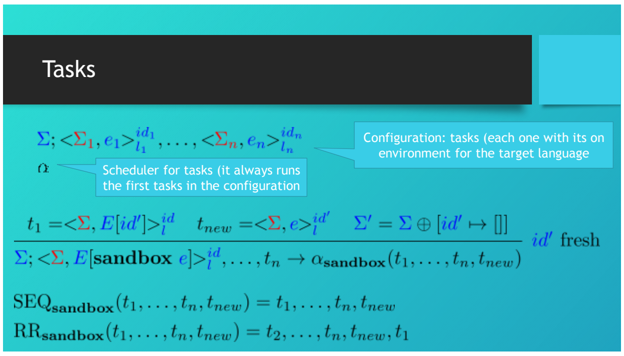
Lifting reduction rules II

Target “outside”, IFC ”inside”

$$\frac{e \neq \lfloor e \rfloor \quad \Sigma, \Sigma, E[e] \to \Sigma, \Sigma', E[e']}{\Sigma, \Sigma, E[e] \to \Sigma, \Sigma', E[e']}$$

$$\frac{e = \lfloor e \rfloor \quad \exists E, e_1 \cdot e = E[e_1] \wedge \Sigma, \Sigma, E[e_1] \to \Sigma', \Sigma, E[e_2]}{\Sigma, \Sigma, E[e] \to \Sigma', \Sigma, E[\lfloor E[e_2] \rfloor]}$$

IFC “outside”, target “inside”? Homework!



Tasks

$\Sigma; <\Sigma_1, e_1>^{id_1}_{l_1}, \ldots, <\Sigma_n, e_n>^{id_n}_{l_n}$

Configuration: tasks (each one with its on environment for the target language

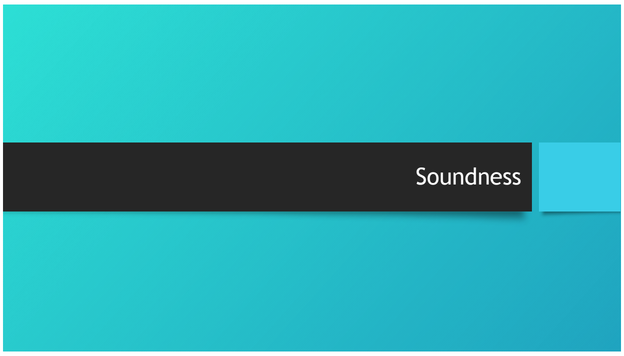
α

Scheduler for tasks (it always runs the first tasks in the configuration

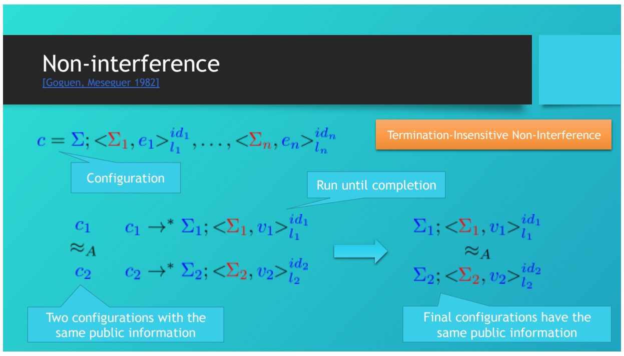
$$\frac{t_1 = <\Sigma, E[id']>^{id}_{l} \quad t_{new} = <\Sigma, e>^{id'}_{l} \quad \Sigma' = \Sigma \oplus [id' \mapsto []]}{\Sigma; <\Sigma, E[\mathbf{sandbox}\ e]>^{id}_{l}, \ldots, t_n \to \alpha_{\mathbf{sandbox}}(t_1, \ldots, t_n, t_{new})}\ id'\ \text{fresh}$$

$\mathrm{SEQ}_{\mathbf{sandbox}}(t_1, \ldots, t_n, t_{new}) = t_1, \ldots, t_n, t_{new}$

$\mathrm{RR}_{\mathbf{sandbox}}(t_1, \ldots, t_n, t_{new}) = t_2, \ldots, t_n, t_{new}, t_1$



Soundness



Non-interference

[Goguen, Meseguer 1982]

$c = \Sigma; <\Sigma_1, e_1>^{id_1}_{l_1}, \ldots, <\Sigma_n, e_n>^{id_n}_{l_n}$

Termination-Insensitive Non-Interference

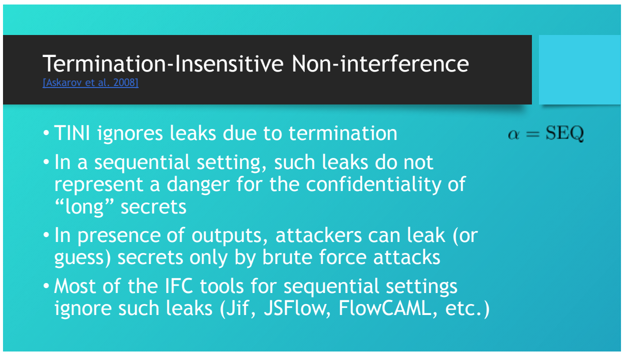
Configuration

Run until completion

$$\begin{array}{c} c_1 \\ \approx_A \\ c_2 \end{array} \quad \begin{array}{l} c_1 \to^* \Sigma_1; <\Sigma_1, v_1>^{id_1}_{l_1} \\ \\ c_2 \to^* \Sigma_2; <\Sigma_2, v_2>^{id_2}_{l_2} \end{array} \quad \Longrightarrow \quad \begin{array}{c} \Sigma_1; <\Sigma_1, v_1>^{id_1}_{l_1} \\ \approx_A \\ \Sigma_2; <\Sigma_2, v_2>^{id_2}_{l_2} \end{array}$$

Two configurations with the same public information

Final configurations have the same public information

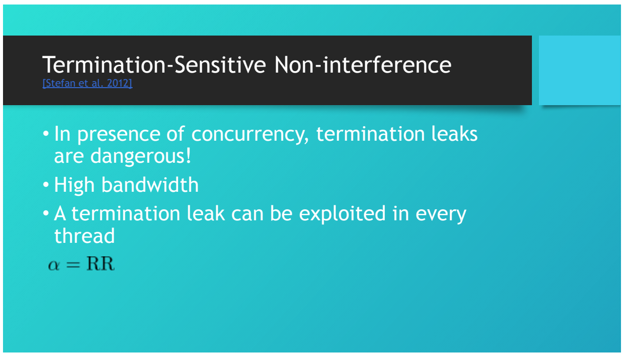


Termination-Insensitive Non-interference

[Askarov et al. 2008]

$\alpha = \mathrm{SEQ}$

- TINI ignores leaks due to termination
- In a sequential setting, such leaks do not represent a danger for the confidentiality of “long” secrets
- In presence of outputs, attackers can leak (or guess) secrets only by brute force attacks
- Most of the IFC tools for sequential settings ignore such leaks (Jif, JSFlow, FlowCAML, etc.)

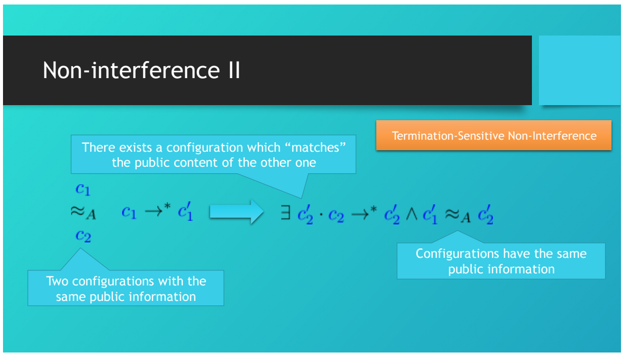


Termination-Sensitive Non-interference

[Stefan et al. 2012]

- In presence of concurrency, termination leaks are dangerous!
- High bandwidth
- A termination leak can be exploited in every thread

$\alpha = \mathrm{RR}$



Non-interference II

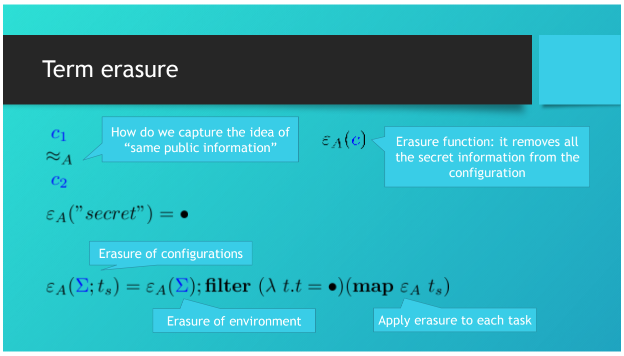
Termination-Sensitive Non-Interference

There exists a configuration which “matches” the public content of the other one

$$\begin{array}{c} c_1 \\ \approx_A \\ c_2 \end{array} \quad c_1 \to^* c'_1 \quad \Longrightarrow \quad \exists\, c'_2 \cdot c_2 \to^* c'_2 \wedge c'_1 \approx_A c'_2$$

Two configurations with the same public information

Configurations have the same public information



Term erasure

$c_1 \approx_A c_2$

How do we capture the idea of "same public information"

$\varepsilon_A(c)$

Erasure function: it removes all the secret information from the configuration

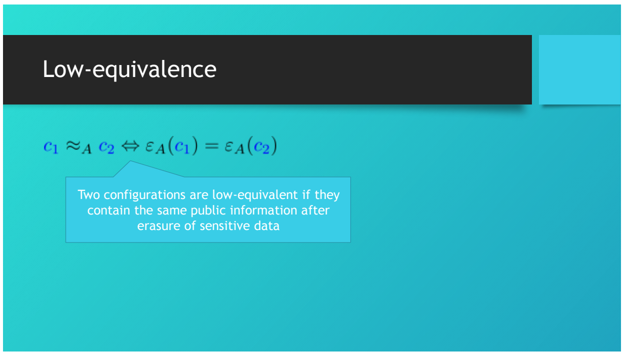
$\varepsilon_A(\text{"}secret\text{"}) = \bullet$

Erasure of configurations

$\varepsilon_A(\Sigma; t_s) = \varepsilon_A(\Sigma); \mathbf{filter}\ (\lambda\ t.t = \bullet)(\mathbf{map}\ \varepsilon_A\ t_s)$

Erasure of environment

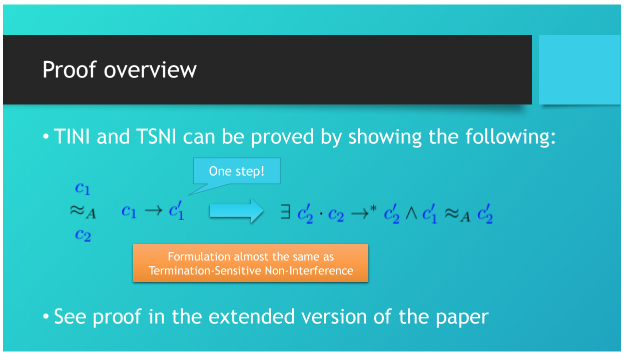
Apply erasure to each task



Low-equivalence

$c_1 \approx_A c_2 \Leftrightarrow \varepsilon_A(c_1) = \varepsilon_A(c_2)$

Two configurations are low-equivalent if they contain the same public information after erasure of sensitive data



Proof overview

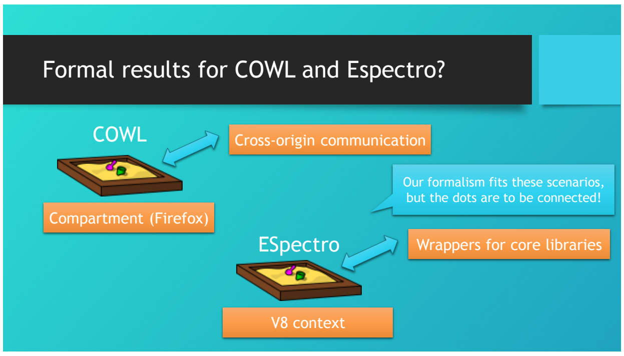
- TINI and TSNI can be proved by showing the following:

$c_1 \approx_A c_2 \quad c_1 \to c_1' \Rightarrow \exists\, c_2' \cdot c_2 \to^* c_2' \wedge c_1' \approx_A c_2'$

One step!

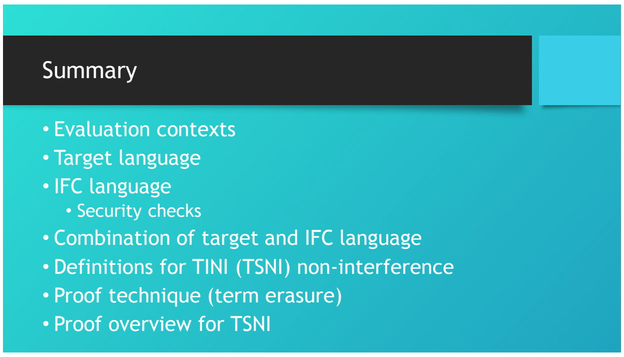
Formulation almost the same as Termination-Sensitive Non-Interference

- See proof in the extended version of the paper



Formal results for COWL and Espectro?

Our formalism fits these scenarios, but the dots are to be connected!



Summary

- Evaluation contexts
- Target language
- IFC language
  - Security checks
- Combination of target and IFC language
- Definitions for TINI (TSNI) non-interference
- Proof technique (term erasure)
- Proof overview for TSNI

IFC Inside: Retrofitting Languages with Dynamic Information Flow Control

Stefan Heule[1], Deian Stefan[1], Edward Z. Yang[1], John C. Mitchell[1], and Alejandro Russo[2]**

[1] Stanford University
[2] Chalmers University

Abstract. Many important security problems in JavaScript, such as browser extension security, untrusted JavaScript libraries and safe integration of mutually distrustful websites (mash-ups), may be effectively addressed using an efficient implementation of information flow control (IFC). Unfortunately existing fine-grained approaches to JavaScript IFC require modifications to the language semantics and its engine, a non-goal for browser applications. In this work, we take the ideas of coarse-grained dynamic IFC and provide the theoretical foundation for a language-based approach that can be applied to any programming language for which external effects can be controlled. We then apply this formalism to server- and client-side JavaScript, show how it generalizes to the C programming language, and connect it to the Haskell LIO system. Our methodology offers design principles for the construction of information flow control systems when isolation can easily be achieved, as well as compositional proofs for optimized concrete implementations of these systems, by relating them to their isolated variants.

1 Introduction

Modern web content is rendered using a potentially large number of different components with differing provenance. Disparate and untrusting components may arise from browser extensions (whose JavaScript code runs alongside website code), web applications (with possibly untrusted third-party libraries), and mashups (which combine code and data from websites that may not even be aware of each other's existence.) While just-in-time combination of untrusting components offers great flexibility, it also poses complex security challenges. In particular, maintaining data privacy in the face of malicious extensions, libraries, and mashup components has been difficult.

Information flow control (IFC) is a promising technique that provides security by tracking the flow of sensitive data through a system. Untrusted code is confined so that it cannot exfiltrate data, except as per an information flow policy. Significant research has been devoted to adding various forms of IFC to different kinds of programming languages and systems. In the context of the web, however, there is a strong motivation to preserve JavaScript's semantics

** Work partially done while at Stanford.

and avoid JavaScript-engine modifications, while retrofitting it with dynamic information flow control.

The Operating Systems community has tackled this challenge (e.g., in [45]) by taking a *coarse-grained* approach to IFC: dividing an application into coarse computational units, each with a single label dictating its security policy, and only monitoring communication between them. This coarse-grained approach provides a number of advantages when compared to the fine-grained approaches typically employed by language-based systems. First, adding IFC does not require intrusive changes to an existing programming language, thereby also allowing the reuse of existing programs. Second, it has a small runtime overhead because checks need only be performed at isolation boundaries instead of (almost) every program instruction (e.g., [19]). Finally, associating a single security label with the entire computational unit simplifies understanding and reasoning about the security guarantees of the system, without reasoning about most of the technical details of the semantics of the underlying programming language.

In this paper, we present a framework which brings coarse-grained IFC ideas into a language-based setting: an information flow control system should be thought of as multiple instances of completely isolated language runtimes or *tasks*, with information flow control applied to inter-task communication. We describe a formal system in which an IFC system can be designed once and then applied to any programming language which has control over external effects (e.g., JavaScript or C with access to hardware privilege separation). We formalize this system using an approach by Matthews and Findler [25] for combining operational semantics and prove non-interference guarantees that are independent of the choice of a specific target language.

There are a number of points that distinguish this setting from previous coarse-grained IFC systems. First, even though the underlying semantic model involves communicating tasks, these tasks can be coordinated together in ways that simulate features of traditional languages. In fact, simulating features in this way is a useful *design tool* for discovering what variants of the features are permissible and which are not. Second, although completely separate tasks are semantically easy to reason about, real-world implementations often blur the lines between tasks in the name of efficiency. Characterizing what optimizations are permissible is subtle, since removing transitions from the operational semantics of a language can break non-interference. We partially address this issue by characterizing isomorphisms between the operational semantics of our abstract language and a concrete implementation, showing that if this relationship holds, then non-interference in the abstract specification carries over to the concrete implementation.

Our contributions can be summarized as follows:

– We give formal semantics for a core coarse-grained dynamic information flow control language free of non-IFC constructs. We then show how a large class of target languages can be combined with this IFC language and prove that the result provides non-interference. (Sections 2 and 3)
– We provide a proof technique to show the non-interference of a concrete semantics for a potentially optimized IFC language by means of an isomor-

phism and show a class of restrictions on the IFC language that preserves non-interference. (Section 4)

- We have implemented an IFC system based on these semantics for Node.js, and we connect our formalism to another implementation based on this work for client-side JavaScript [37]. Furthermore, we outline an implementation for the C programming language and describe improvements to the Haskell LIO system that resulted from this framework. (Section 5)

In the extended version of this paper we give all the relevant proofs and extend our IFC language with additional features [20].

2 Retrofitting Languages with IFC

Before moving on to the formal treatment of our system, we give a brief primer of information flow control and describe some example programs in our system, emphasizing the parallel between their implementation in a multi-task setting, and the traditional, "monolithic" programming language feature they simulate.

Information flow control systems operate by associating data with *labels*, and specifying whether or not data tagged with one label l_1 can flow to another label l_2 (written as $l_1 \sqsubseteq l_2$). These labels encode the desired security policy (for example, confidential information should not flow to a public channel), while the work of specifying the semantics of an information flow language involves demonstrating that impermissible flows cannot happen, a property called *non-interference* [17]. In our coarse-grained floating-label approach, labels are associated with tasks. The task label—we refer to the label of the currently executing task as the *current label*—serves to protect everything in the task's scope; all data in a task shares this common label.

As an example, here is a program which spawns a new isolated task, and then sends it a mutable reference:

$$\begin{array}{l}\textbf{let}\ i = {}_{\mathrm{TI}}\lfloor \textbf{sandbox}\ (\textbf{blockingRecv}\ x, \_\ \textbf{in}\ {}^{\mathrm{IT}}\lceil\, !\ {}_{\mathrm{TI}}\lfloor x \rfloor \rceil) \rfloor \\ \textbf{in}\ {}_{\mathrm{TI}}\lfloor \textbf{send}\ {}^{\mathrm{IT}}\lceil i \rceil\ l\ {}^{\mathrm{IT}}\lceil \textbf{ref true} \rceil \rfloor\end{array}$$

For now, ignore the tags ${}_{\mathrm{TI}}\lfloor \cdot \rfloor$ and ${}^{\mathrm{IT}}\lceil \cdot \rceil$: roughly, this code creates a new **sandbox**ed task with identifier i which waits (**blockingRecv**, binding x with the received message) for a message, and then **send**s the task a mutable reference (**ref true**) which it labels l. If this operation actually shared the mutable cell between the two tasks, it could be used to violate information flow control if the tasks had differing labels. At this point, the designer of an IFC system might add label checks to mutable references, to check the labels of the reader and writer. While this solves the leak, for languages like JavaScript, where references are prevalently used, this also dooms the performance of the system.

Our design principles suggest a different resolution: when these constructs are treated as isolated tasks, each of which have their own heaps, it is obviously the case that there is no sharing; in fact, the sandboxed task receives a dangling pointer. Even if there is only one heap, if we enforce that references not be shared, the two systems are morally equivalent. (We elaborate on this formally in Section 4.) Finally, this semantics strongly suggests that one should restrict the types of data which may be passed between tasks (for example, in JavaScript, one might only allow JSON objects to be passed between tasks, rather than general object structures).

Existing language-based, coarse-grained IFC systems [21, 35] allow a sub-computation to temporarily raise the floating-label; after the sub-computation is done, the floating-label is restored to its original label. When this occurs, the enforcement mechanism must ensure that information does not leak to the (less confidential) program continuation. The presence of exceptions adds yet more intricacies. For instance, exceptions should not automatically propagate from a sub-computation directly into the program continuation, and, if such exceptions are allowed to be inspected, the floating-label at the point of the exception-raise must be tracked alongside the exception value [18, 21, 35]. In contrast, our system provides the same flexibility and guarantees with no extra checks: tasks are used to execute sub-computations, but the mere definition of isolated tasks guarantees that (a) tasks only transfer data to the program continuation by using inter-task communication means, and (b) exceptions do cross tasks boundaries automatically.

2.1 Preliminaries

Our goal now is to describe how to take a **target language** with a formal operational semantics and combine it with an *information flow control language*. For example, taking ECMAScript as the target language and combining it with our IFC language should produce the formal semantics for the core part of COWL [37]. In this presentation, we use a simple, untyped lambda calculus with mutable references and fixpoint in place of ECMAScript to demonstrate some the key properties of the system (and, because the embedding does not care about the target language features); we discuss the proper embedding in more detail in Section 5.

Notation We have typeset nonterminals of the target language using **bold font** while the nonterminals of the IFC language have been typeset with *italic font*. Readers are encouraged to view a color copy of this paper, where target language nonterminals are colored **red** and IFC language nonterminals are colored *blue*.

2.2 Target Language: Mini-ES

In Fig. 1, we give a simple, untyped lambda calculus with mutable references and fixpoint, prepared for combination with an information flow control language. The presentation is mostly standard, and utilizes Felleisen-Hieb reduction semantics [16] to define the operational semantics of the system. One peculiarity is that our language defines an evaluation context $\mathbf{E}$, but, the evaluation rules have been expressed in terms of a different evaluation context $\mathcal{E}_{\mathbf{\Sigma}}$; Here, we follow the approach of Matthews and Findler [25] in order to simplify combining semantics of multiple languages. To derive the usual operational semantics for this language, the evaluation context merely needs to be defined as $\mathcal{E}_{\mathbf{\Sigma}}[\mathbf{e}] \triangleq \mathbf{\Sigma}, \mathbf{E}[\mathbf{e}]$. However, when we combine this language with an IFC language, we reinterpret the meaning of this evaluation context.

$$\begin{array}{lcl}
\mathbf{v} & ::= & \lambda \mathbf{x}.\mathbf{e} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{a} \\
\mathbf{e} & ::= & \mathbf{v} \mid \mathbf{x} \mid \mathbf{e}\ \mathbf{e} \mid \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}\ \mathbf{else}\ \mathbf{e} \mid \mathbf{ref}\ \mathbf{e} \mid !\mathbf{e} \mid \mathbf{e} := \mathbf{e} \mid \mathbf{fix}\ \mathbf{e} \\
\mathbf{E} & ::= & [\cdot]_{\mathbf{T}} \mid \mathbf{E}\ \mathbf{e} \mid \mathbf{v}\ \mathbf{E} \mid \mathbf{if}\ \mathbf{E}\ \mathbf{then}\ \mathbf{e}\ \mathbf{else}\ \mathbf{e} \mid \mathbf{ref}\ \mathbf{E} \mid !\mathbf{E} \mid \mathbf{E} := \mathbf{e} \mid \mathbf{v} := \mathbf{E} \mid \mathbf{fix}\ \mathbf{E}
\end{array}$$

$$\begin{array}{ll}
\mathbf{e}_1 ; \mathbf{e}_2 & \triangleq (\lambda \mathbf{x}.\mathbf{e}_2)\ \mathbf{e}_1\ \mathbf{where}\ \mathbf{x} \notin \mathcal{FV}\ (\mathbf{e}_2) \\
\mathbf{let}\ \mathbf{x} = \mathbf{e}_1\ \mathbf{in}\ \mathbf{e}_2 & \triangleq (\lambda \mathbf{x}.\mathbf{e}_2)\ \mathbf{e}_1
\end{array}$$

$$\textsc{T-app}\quad \frac{}{\mathcal{E}_{\mathbf{\Sigma}}\ [(\lambda x.\mathbf{e})\ \mathbf{v}] \rightarrow \mathcal{E}_{\mathbf{\Sigma}}\ [\{\mathbf{v}\ /\ x\}\ \mathbf{e}]} \qquad \textsc{T-ifTrue}\quad \frac{}{\mathcal{E}_{\mathbf{\Sigma}}\ [\ \mathbf{if}\ \mathbf{true}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2] \rightarrow \mathcal{E}_{\mathbf{\Sigma}}\ [\mathbf{e}_1]}$$

Fig. 1: λ_{ES}: simple untyped lambda calculus extended with booleans, mutable references and general recursion. For space reasons we only show two representative reduction rules; full rules can be found in the extended version of this paper.

In general, we require that a target language be expressed in terms of some global machine state $\mathbf{\Sigma}$, some evaluation context $\mathbf{E}$, some expressions $\mathbf{e}$, some set of values $\mathbf{v}$ and a *deterministic* reduction relation on full configurations $\mathbf{\Sigma} \times \mathbf{E} \times \mathbf{e}$.

2.3 IFC Language

As mentioned previously, most modern, dynamic information flow control languages encode policy by associating a label with data. Our embedding is agnostic to the choice of labeling scheme; we only require the labels to form a lattice [12] with the partial order $\sqsubseteq$, join $\sqcup$, and meet $\sqcap$. In this paper, we simply represent labels with the metavariable l, but do not discuss them in more detail. To enforce labels, the IFC monitor inspects the current label before performing a read or a write to decide whether the operation is permitted. A task can only write to entities that are at least as sensitive. Similarly, it can only read from entities that are less sensitive. However, as in other floating-label systems, this current label can be raised to allow the task to read from more sensitive entities at the cost of giving up the ability to write to others.

In Fig. 2, we give the syntax and *single-task* evaluation rules for a minimal information flow control language. Ordinarily, information flow control languages are defined by directly stating a base language plus information flow control operators. In contrast, our language is purposely minimal: it does not have sequencing operations, control flow, or other constructs. However, it contains support for the following core information flow control features:

- First-class labels, with label values l as well as operations for computing on labels ($\sqsubseteq$, $\sqcup$ and $\sqcap$).
- Operations for inspecting (**getLabel**) and modifying (**setLabel**) the current label of the task (a task can only increase its label).
- Operations for non-blocking inter-task communication (**send** and **recv**), which interact with the global store of per-task message queues Σ.
- A sandboxing operation used to spawn new isolated tasks. In concurrent settings **sandbox** corresponds to a fork-like primitive, whereas in a sequential setting, it more closely resembles computations which might temporarely raise the current floating-label [21, 33].

These operations are all defined with respect to an evaluation context $\mathcal{E}_{\Sigma}^{i,l}$ that represents the context of the current task. The evaluation context has three important pieces of state: the global message queues Σ, the current label l and the task ID i.

We note that first-class labels, tasks (albeit named differently), and operations for inspecting the current label are essentially universal to all floating-label systems. However, our choice of communication primitives is motivated by those present in browsers, namely `postMessage` [41]. Of course, other choices, such as blocking communication or labeled channels, are possible.

These asynchronous communication primitives are worth further discussion. When a task is sending a message using **send**, it also labels that message with a label l' (which must be at or above the task's current label l). Messages can only be received by a task if its current label is at least as high as the label of the message. Specifically, receiving a message using **recv** x_1, x_2 **in** e_1 **else** e_2 binds the message and the sender's task identifier to local variables x_1 and x_2, respectively, and then executes e_1. Otherwise, if there are no messages, that task continues its execution with e_2. We denote the filtering of the message queue by $\Theta \preceq l$, which is defined as follows. If Θ is the empty list **nil**, the function is simply the identity function, i.e., $\textbf{nil} \preceq l = \textbf{nil}$, and otherwise:

$$((l', i, e), \Theta) \preceq l = \begin{cases} (l', i, e), (\Theta \preceq l) & \text{if } l' \sqsubseteq l \\ \Theta \preceq l & \text{otherwise} \end{cases}$$

This ensures that tasks cannot receive messages that are more sensitive than their current label would allow.

2.4 The Embedding

Fig. 3 provides all of the rules responsible for actually carrying out the embedding of the IFC language within the target language. The most important feature of this embedding is that every task maintains its own copy of the target language global state and evaluation context, thus enforcing isolation between various tasks. In more detail:

- We extend the values, expressions and evaluation contexts of both languages to allow for terms in one language to be embedded in the other, as in [25]. In the target language, an IFC expression appears as $_{\mathrm{TI}}\lfloor e \rfloor$ ("Target-outside, IFC-inside"); in the IFC language, a target language expression appears as $^{\mathrm{IT}}\lceil \mathbf{e} \rceil$ ("IFC-outside, target-inside").
- We reinterpret $\mathcal{E}$ to be evaluation contexts on task lists, providing definitions for $\mathcal{E}_{\mathbf{\Sigma}}$ and $\mathcal{E}_{\Sigma}^{i,l}$. These rules only operate on the first task in the task list, which by convention is the only task executing.
- We reinterpret $\rightarrow$, an operation on a single task, in terms of $\hookrightarrow$, operation on task lists. The correspondence is simple: a task executes a step and then is rescheduled in the task list according to schedule policy α. Fig. 4 defines two concrete schedulers.
- Finally, we define some rules for scheduling, handling sandboxing tasks (which interact with the state of the target language), and intermediating between the borders of the two languages.

$$\begin{aligned}
v &::= i \mid l \mid \mathbf{true} \mid \mathbf{false} \mid \langle\rangle \qquad\qquad \otimes ::= \sqsubseteq \mid \sqcup \mid \sqcap \\
e &::= v \mid x \mid e \otimes e \mid \mathbf{getLabel} \mid \mathbf{setLabel}\ e \mid \mathbf{taskId} \mid \mathbf{sandbox}\ e \\
&\quad\ \mid \mathbf{send}\ e\ e\ e \mid \mathbf{recv}\ x, x\ \mathbf{in}\ e\ \mathbf{else}\ e \\
E &::= [\cdot]_I \mid E \otimes e \mid v \otimes E \mid \mathbf{setLabel}\ E \mid \mathbf{send}\ E\ e\ e \mid \mathbf{send}\ v\ E\ e \mid \mathbf{send}\ v\ v\ E \\
\theta &::= (l, i\ e) \qquad\qquad \Theta ::= \mathbf{nil} \mid \theta, \Theta \qquad\qquad \Sigma ::= \emptyset \mid \Sigma\,[i \mapsto \Theta]
\end{aligned}$$

I-getTaskId
$$\frac{}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{taskId}] \to \mathcal{E}^{i,l}_{\Sigma}\,[i]}$$

I-getLabel
$$\frac{}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{getLabel}] \to \mathcal{E}^{i,l}_{\Sigma}\,[l]}$$

I-labelOp
$$\frac{[\![l_1 \otimes l_2]\!] = v}{\mathcal{E}^{i,l}_{\Sigma}\,[l_1 \otimes l_2] \to \mathcal{E}^{i,l}_{\Sigma}\,[v]}$$

I-send
$$\frac{l \sqsubseteq l' \qquad \Sigma(i') = \Theta \qquad \Sigma' = \Sigma\left[i' \mapsto (l', i, v), \Theta\right]}{\mathcal{E}^{i,l}_{\Sigma}\,\left[\mathbf{send}\ i'\ l'\ v\right] \to \mathcal{E}^{i,l}_{\Sigma'}\,[\langle\rangle]}$$

I-recv
$$\frac{(\Sigma(i) \preceq l) = \theta_1, ..., \theta_k, (l', i', v) \qquad \Sigma' = \Sigma\,[i \mapsto (\theta_1, ..., \theta_k)]}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{recv}\ x_1, x_2\ \mathbf{in}\ e_1\ \mathbf{else}\ e_2] \to \mathcal{E}^{i,l}_{\Sigma'}\,\left[\{v \,/\, x_1, i' \,/\, x_2\}\ e_1\right]}$$

I-noRecv
$$\frac{\Sigma(i) \preceq l = \mathbf{nil} \qquad \Sigma' = \Sigma\,[i \mapsto \mathbf{nil}]}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{recv}\ x_1, x_2\ \mathbf{in}\ e_1\ \mathbf{else}\ e_2] \to \mathcal{E}^{i,l}_{\Sigma'}\,[e_2]}$$

I-setLabel
$$\frac{l \sqsubseteq l'}{\mathcal{E}^{i,l}_{\Sigma}\,\left[\mathbf{setLabel}\ l'\right] \to \mathcal{E}^{i,l'}_{\Sigma}\,[\langle\rangle]}$$

Fig. 2: IFC language with all single-task operations.

$$\begin{aligned}
v &::= \cdots \mid {}^{\mathrm{IT}}\lceil \mathbf{v} \rceil & \mathbf{v} &::= \cdots \mid {}_{\mathrm{TI}}\lfloor v \rfloor & \mathcal{E}_{\boldsymbol{\Sigma}}\,[\mathbf{e}] &\triangleq \Sigma; \langle \boldsymbol{\Sigma}, E[\mathbf{e}]_{\mathbf{T}} \rangle^i_l, \ldots \\
e &::= \cdots \mid {}^{\mathrm{IT}}\lceil \mathbf{e} \rceil & \mathbf{e} &::= \cdots \mid {}_{\mathrm{TI}}\lfloor e \rfloor & \mathcal{E}^{i,l}_{\Sigma}\,[e] &\triangleq \Sigma; \langle \boldsymbol{\Sigma}, E[e]_I \rangle^i_l, \ldots \\
E &::= \cdots \mid {}^{\mathrm{IT}}\lceil \mathbf{E} \rceil & \mathbf{E} &::= \cdots \mid {}_{\mathrm{TI}}\lfloor E \rfloor & \mathcal{E}\,[e] \to \Sigma; t, \ldots &\triangleq \mathcal{E}\,[e] \overset{\alpha}{\hookrightarrow} \Sigma; \alpha_{\mathrm{step}}(t, \ldots)
\end{aligned}$$

I-sandbox
$$\frac{\begin{array}{c}\Sigma' = \Sigma\left[i' \mapsto \mathbf{nil}\right] \\ \boldsymbol{\Sigma}' = \kappa\,(\boldsymbol{\Sigma}) \qquad t_1 = \langle \boldsymbol{\Sigma}, E[i'] \rangle^i_l \qquad t_{\mathrm{new}} = \langle \boldsymbol{\Sigma}', e \rangle^{i'}_l \qquad \mathrm{fresh}(i')\end{array}}{\Sigma; \langle \boldsymbol{\Sigma}, E[\mathbf{sandbox}\ e]_I \rangle^i_l, \ldots \overset{\alpha}{\hookrightarrow} \Sigma'; \alpha_{\mathrm{sandbox}}(t_1, \ldots, t_{\mathrm{new}})}$$

I-done
$$\frac{}{\Sigma; \langle \boldsymbol{\Sigma}, v \rangle^i_l, \ldots \overset{\alpha}{\hookrightarrow} \Sigma; \alpha_{\mathrm{done}}(\langle \boldsymbol{\Sigma}, v \rangle^i_l, \ldots)}$$

I-noStep
$$\frac{\Sigma; t, \ldots \overset{\alpha}{\not\hookrightarrow}}{\Sigma; t, \ldots \overset{\alpha}{\hookrightarrow} \Sigma; \alpha_{\mathrm{noStep}}(t, \ldots)}$$

I-border
$$\frac{}{\mathcal{E}^{i,l}_{\Sigma}\left[{}^{\mathrm{IT}}\lceil {}_{\mathrm{TI}}\lfloor e \rfloor \rceil\right] \to \mathcal{E}^{i,l}_{\Sigma}\,[e]}$$

T-border
$$\frac{}{\mathcal{E}_{\boldsymbol{\Sigma}}\left[{}_{\mathrm{TI}}\lfloor {}^{\mathrm{IT}}\lceil \mathbf{e} \rceil \rfloor\right] \to \mathcal{E}_{\boldsymbol{\Sigma}}\,[\mathbf{e}]}$$

Fig. 3: The embedding $L_{\mathrm{IFC}}(\alpha, \lambda)$, where $\lambda = (\boldsymbol{\Sigma}, \mathbf{E}, \mathbf{e}, \mathbf{v}, \to)$

$$\begin{aligned}
\mathrm{RR}_{\mathrm{step}}(t_1, t_2, \ldots) &= t_2, \ldots, t_1 \\
\mathrm{RR}_{\mathrm{done}}(t_1, t_2, \ldots) &= t_2, \ldots \\
\mathrm{RR}_{\mathrm{noStep}}(t_1, t_2, \ldots) &= t_2, \ldots \\
\mathrm{RR}_{\mathrm{sandbox}}(t_1, t_2, \ldots) &= t_2, \ldots, t_1
\end{aligned}
\qquad
\begin{aligned}
\mathrm{SEQ}_{\mathrm{step}}(t_1, t_2, \ldots) &= t_1, t_2, \ldots \\
\mathrm{SEQ}_{\mathrm{noStep}}(t_1, t_2, \ldots) &= t_1, t_2, \ldots \\
\mathrm{SEQ}_{\mathrm{done}}(t) &= t \\
\mathrm{SEQ}_{\mathrm{done}}(t_1, t_2, \ldots) &= t_2, \ldots \\
\mathrm{SEQ}_{\mathrm{sandbox}}(t_1, t_2, \ldots, t_n) &= t_n, t_1, t_2, \ldots
\end{aligned}$$

Fig. 4: Scheduling policies (concurrent round robin on the left, sequential on the right).

The I-SANDBOX rule is used to create a new isolated task that executes separately from the existing tasks (and can be communicated with via **send** and **recv**). When the new task is created, there is the question of what the target language state of the new task should be. Our rule is stated generically in terms of a function κ. Conservatively, κ may be simply thought of as the identity function, in which case the semantics of **sandbox** are such that the state of the target language is *cloned* when sandboxing occurs. However, this is not necessary: it is also valid for κ to remove entries from the state. In Section 4, we give a more detailed discussion of the implications of the choice of κ, but all our security claims will hold regardless of the choice of κ.

The rule I-NOSTEP says something about configurations for which it is not possible to take a transition. The notation $c \overset{\alpha}{\not\to}$ in the premise is meant to be understood as follows: If the configuration c cannot take a step by any rule other than I-NOSTEP, then I-NOSTEP applies and the stuck task gets removed.

Rules I-DONE and I-NOSTEP define the behavior of the system when the current thread has reduced to a value, or gotten stuck, respectively. While these definitions simply rely on the underlying scheduling policy α to modify the task list, as we describe in Sections 3 and 6, these rules (notably, I-NOSTEP) are crucial to proving our security guarantees. For instance, it is unsafe for the whole system to get stuck if a particular task gets stuck, since a sensitive thread may then leverage this to leak information through the termination channel. Instead, as our example round-robin (RR) scheduler shows, such tasks should simply be removed from the task list. Many language runtime or Operating System schedulers implement such schedulers. Moreover, techniques such as instruction-based scheduling [10, 36] can be further applied close the gap between specified semantics and implementation.

As in [25], rules T-BORDER and I-BORDER define the syntactic boundaries between the IFC and target languages. Intuitively, the boundaries respectively correspond to an upcall into and downcall from the IFC runtime. As an example, taking λ_{ES} as the target language, we can now define a blocking receive (inefficiently) in terms of the asynchronous **recv** as series of cross-language calls:

$$\textbf{blockingRecv}\ x_1, x_2\ \textbf{in}\ e \triangleq {}^{\mathrm{IT}}\lceil \textbf{fix}\ (\lambda k.{}_{\mathrm{TI}}\lfloor \textbf{recv}\ x_1, x_2\ \textbf{in}\ e\ \textbf{else}\ {}^{\mathrm{IT}}\lceil k \rceil \rfloor) \rceil$$

For any target language λ and scheduling policy α, this embedding defines an IFC language, which we will refer to as $L_{\mathrm{IFC}}(\alpha, \lambda)$.

3 Security Guarantees

We are interested in proving non-interference about many programming languages. This requires an appropriate definition of this notion that is language agnostic, so in this section, we present a few general definitions for what an information flow control language is and what non-interference properties it may have. In particular, we show that $L_{\text{IFC}}(\alpha, \lambda)$, with an appropriate scheduler α, satisfies non-interference [17], without making any reference to properties of λ. We state the appropriate theorems here, and provide the formal proofs in the extended version of this paper.

3.1 Erasure Function

When defining the security guarantees of an information flow control, we must characterize what the *secret inputs* of a program are. Like other work [24, 30, 33, 34], we specify and prove non-interference using *term erasure*. Intuitively, term erasure allows us to show that an attacker does not learn any sensitive information from a program if the program behaves identically (from the attackers point of view) to a program with all sensitive data "erased". To interpret a language under information flow control, we define a function ε_l that performs erasures by mapping configurations to erased configurations, usually by rewriting (parts of) configurations that are more sensitive than l to a new syntactic construct $\bullet$. We define an information flow control language as follows:

Definition 1 (Information flow control language). *An information flow control language* $\boldsymbol{L}$ *is a tuple* $(\Delta, \hookrightarrow, \varepsilon_l)$*, where* Δ *is the type of machine configurations (members of which are usually denoted by the metavariable* c*),* $\hookrightarrow$ *is a reduction relation between machine configurations and* $\varepsilon_l : \Delta \to \varepsilon(\Delta)$ *is an erasure function parametrized on labels from machine configurations to* erased *machine configurations* $\varepsilon(\Delta)$*. Sometimes, we use* V *to refer to set of terminal configurations in* Δ*, i.e., configurations where no further transitions are possible.*

Our language $L_{\text{IFC}}(\alpha, \lambda)$ fulfills this definition as $(\Delta, \overset{\alpha}{\hookrightarrow}, \varepsilon_l)$, where $\Delta = \Sigma \times \text{List}(t)$. The set of terminal conditions V is $\Sigma \times t_V$, where $t_V \subset t$ is the type for tasks whose expressions have been reduced to values.[3] The erased configuration $\varepsilon(\Delta)$ extends Δ with configurations containing $\bullet$, and Fig. 5 gives the precise definition for our erasure function ε_l. Essentially, a task and its corresponding message queue is completely erased from the task list if its label does not flow to the attacker observation level l. Otherwise, we apply the erasure function homomorphically and remove any messages from the task's message queue that are more sensitive than l.

The definition of an erasure function is quite important: it captures the attacker model, stating what can and cannot be observed by the attacker. In our case, we assume that the attacker cannot observe sensitive tasks or messages, or even the number of such entities. While such assumptions are standard [8, 34], our definitions allow for stronger attackers that may be able to inspect resource usage.[4]

[3] Here, we abuse notation by describing types for configuration parts using the same metavariables as the "instance" of the type, e.g., t for the type of task.

[4] We believe that we can extend $L_{\text{IFC}}(\alpha, \lambda)$ to such models using the resource limits techniques of [42]. We leave this extension to future work.

$$
\begin{aligned}
&\varepsilon_l(\Sigma; ts) = \varepsilon_l(\Sigma); \text{filter } (\lambda t.t = \bullet)\ (\text{map } \varepsilon_l\ ts)\\
&\varepsilon_l(\langle \boldsymbol{\Sigma}, e\rangle^{i}_{l'}) = \begin{cases} \bullet & l' \not\sqsubseteq l \\ \langle \varepsilon_l(\boldsymbol{\Sigma}), \varepsilon_l(e)\rangle^{i}_{l'} & \text{otherwise} \end{cases}\\
&\varepsilon_l(\Sigma\,[i \mapsto \Theta]) = \begin{cases} \varepsilon_l(\Sigma) & l' \not\sqsubseteq l, \text{ where } l' \text{ is the label of thread } i \\ \varepsilon_l(\Sigma)\,[i \mapsto \varepsilon_l(\Theta)] & \text{otherwise} \end{cases}\\
&\varepsilon_l(\Theta) = \Theta \preceq l \qquad\qquad\qquad \varepsilon_l(\emptyset) = \emptyset
\end{aligned}
$$

Fig. 5: Erasure function for tasks, queue maps, message queues, and configurations. In all other cases, including target-language constructs, ε_l is applied homomorphically. Note that $\varepsilon_l(e)$ is always equal to e (and similar for $\boldsymbol{\Sigma}$) in this simple setting. However, when the IFC language is extended with more constructs as shown in Section 6, then this will no longer be the case.

3.2 Non-Interference

Given an information flow control language, we can now define non-interference. Intuitively, we want to make statements about the attacker's observational power at some security level l. This is done by defining an equivalence relation called l-equivalence on configurations: an attacker should not be able to distinguish two configurations that are l-equivalent. Since our erasure function captures what an attacker can or cannot observe, we simply define this equivalence as the syntactic-equivalence of erased configurations [34].

Definition 2 (l-equivalence). *In a language $(\Delta, \hookrightarrow, \varepsilon_l)$, two machine configurations $c, c' \in \Delta$ are considered l-equivalent, written as $c \approx_l c'$, if $\varepsilon_l(c) = \varepsilon_l(c')$.*

We can now state that a language satisfies non-interference if an attacker at level l cannot distinguish the runs of any two l-equivalent configurations. This particular property is called termination sensitive non-interference (TSNI). Besides the obvious requirement to not leak secret information to public channels, this definition also requires the termination of public tasks to be independent of secret tasks. Formally, we define TSNI as follows:

Definition 3 (Termination Sensitive Non-Interference (TSNI)). *A language $(\Delta, \hookrightarrow, \varepsilon_l)$ satisfies termination sensitive non-interference if for any label l, and configurations $c_1, c_1', c_2 \in \Delta$, if*

$$c_1 \approx_l c_2 \qquad \textit{and} \qquad c_1 \hookrightarrow^* c_1' \tag{1}$$

then there exists a configuration $c_2' \in \Delta$ such that

$$c_1' \approx_l c_2' \qquad \textit{and} \qquad c_2 \hookrightarrow^* c_2'\ . \tag{2}$$

In other words, if we take two l-equivalent configurations, then for every intermediate step taken by the first configuration, there is a corresponding number of steps that the second configuration can take to result in a configuration that is l-equivalent to the first resultant configuration. By symmetry, this applies to all intermediate steps from the second configuration as well.

Our language satisfies TSNI under the round-robin scheduler RR of Fig. 4.

Theorem 1 (Concurrent IFC language is TSNI). *For any target language* λ, $L_{IFC}(\mathrm{RR}, \lambda)$ *satisfies TSNI.*

In general, however, non-interference will not hold for an arbitrary scheduler α. For example, $L_{\mathrm{IFC}}(\alpha, \lambda)$ with a scheduler that inspects a sensitive task's current state when deciding which task to schedule next will in general break non-interference [4, 29].

However, even non-adversarial schedulers are not always safe. Consider, for example, the sequential scheduling policy Seq given in Fig. 4. It is easy to show that $L_{\mathrm{IFC}}(\textsc{Seq}, \lambda)$ does not satisfy TSNI: consider a target language similar to λ_{ES} with an additional expression terminal $\Uparrow$ that denotes a divergent computation, i.e., $\Uparrow$ always reduces to $\Uparrow$ and a simple label lattice $\{\mathsf{pub}, \mathsf{sec}\}$ such that $\mathsf{pub} \sqsubseteq \mathsf{sec}$, but $\mathsf{sec} \not\sqsubseteq \mathsf{pub}$. Consider the following two configurations in this language:

$$c_1 = \Sigma; \langle \mathbf{\Sigma}_1, {}^{\mathrm{IT}}\lceil\ \textbf{if false then} \Uparrow \textbf{else true}\rceil\rangle^1_{\mathsf{sec}}, \langle \mathbf{\Sigma}_2, e\rangle^2_{\mathsf{pub}}$$
$$c_2 = \Sigma; \langle \mathbf{\Sigma}_1, {}^{\mathrm{IT}}\lceil\ \textbf{if true then} \Uparrow \textbf{else true}\rceil\rangle^1_{\mathsf{sec}}, \langle \mathbf{\Sigma}_2, e\rangle^2_{\mathsf{pub}}$$

These two configurations are pub-equivalent, but c_1 will reduce (in two steps) to $c_1' = \Sigma; \langle \mathbf{\Sigma}_1, {}^{\mathrm{IT}}\lceil\textbf{true}\rceil\rangle^2_{\mathsf{pub}}$, whereas c_2 will not make any progress. Suppose that e is a computation that writes to a pub channel,[5] then the sec task's decision to diverge or not is directly leaked to a public entity.

To accommodate for sequential languages, or cases where a weaker guarantee is sufficient, we consider an alternative non-interference property called termination insensitive non-interference (TINI). This property can also be upheld by sequential languages at the cost of leaking through (non)-termination [3].

Definition 4 (Termination insensitive non-interference (TINI)). *A language* $(\Delta, V, \hookrightarrow, \varepsilon_l)$ *is termination insensitive non-interfering if for any label* l, *and configurations* $c_1, c_2 \in \Delta$ *and* $c_1', c_2' \in V$, *it holds that*

$$(c_1 \approx_l c_2 \quad \wedge \quad c_1 \hookrightarrow^* c_1' \quad \wedge \quad c_2 \hookrightarrow^* c_2') \implies c_1' \approx_l c_2'$$

TINI states that if we take two l-equivalent configurations, and both configurations reduce to final configurations (i.e., configurations for which there are no possible further transitions), then the end configurations are also l-equivalent. We highlight that this statement is much weaker than TSNI: it only states that terminating programs do not leak sensitive data, but makes no statement about non-terminating programs.

As shown by compilers [26, 31], interpreters [19], and libraries [30, 33], TINI is useful for sequential settings. In our case, we show that our IFC language with the sequential scheduling policy Seq satisfies TINI.

Theorem 2 (Sequential IFC language is TINI). *For any target language* λ, $L_{IFC}(\textsc{Seq}, \lambda)$ *satisfies TINI.*

[5] Though we do not model labeled channels, extending the calculus with such a feature is straightforward, see Section 6.

4 Isomorphisms and Restrictions

The operational semantics we have defined in the previous section satisfy non-interference by design. We achieve this general statement that works for a large class of languages by having different tasks executing completely isolated from each other, such that every task has its own state. In some cases, this strong separation is desirable, or even necessary. Languages like C provide direct access to memory locations without mechanisms in the language to achieve a separation of the heap. On the other hand, for other languages, this strong isolation of tasks can be undesirable, e.g., for performance reasons. For instance, for the language λ_{ES}, our presentation so far requires a separate heap per task, which is not very practical. Instead, we would like to more tightly couple the integration of the target and IFC languages by reusing existing infrastructure. In the running example, a concrete implementation might use a single global heap. More precisely, instead of using a configuration of the form $\Sigma; \langle \mathbf{\Sigma}_1, e_1\rangle^{i_1}_{l_1}, \langle \mathbf{\Sigma}_2, e_2\rangle^{i_2}_{l_2} \ \ldots$ we would like a single global heap as in $\Sigma; \mathbf{\Sigma}; \langle e_1\rangle^{i_1}_{l_1}, \langle e_2\rangle^{i_2}_{l_2}, \ldots$

If the operational rules are adapted naïvely to this new setting, then non-interference can be violated: as we mentioned earlier, shared mutable cells could be used to leak sensitive information. What we would like is a way of characterizing safe modifications to the semantics which preserve non-interference. The intention of our single heap implementation is to permit efficient execution while *conceptually maintaining isolation between tasks* (by not allowing sharing of references between them). This intuition of having a different (potentially more efficient) concrete semantics that behaves like the abstract semantics can be formalized by the following definition:

Definition 5 (Isomorphism of information flow control languages). *A language $(\Delta, \hookrightarrow, \varepsilon_l)$ is isomorphic to a language $(\Delta', \hookrightarrow', \varepsilon'_l)$ if there exist total functions $f : \Delta \to \Delta'$ and $f^{-1} : \Delta' \to \Delta$ such that $f \circ f^{-1} = id_\Delta$ and $f^{-1} \circ f = id_{\Delta'}$. Furthermore, f and f^{-1} are functorial (e.g., if x' R' y' then $f(x')$ R $f(y')$) over both l-equivalences and $\hookrightarrow$.*

If we weaken this restriction such that f^{-1} does not have to be functorial over $\hookrightarrow$, we call the language $(\Delta, \hookrightarrow, \varepsilon_l)$ weakly isomorphic to $(\Delta', \hookrightarrow', \varepsilon'_l)$.

Providing an isomorphism between the two languages allows us to preserve (termination sensitive or insensitive) non-interference as the following two theorems state.

Theorem 3 (Isomorphism preserves TSNI). *If L is isomorphic to L' and L' satisfies TSNI, then L satisfies TSNI.*

Proof. Shown by transporting configurations and reduction derivations from L to L', applying TSNI, and then transporting the resulting configuration, l-equivalence and multi-step derivation back. □

Only weak isomorphism is necessary for TINI. Intuitively, this is because it is not necessary to back-translate reduction sequences in L' to L; by the definition of TINI, we have both reduction sequences in L by assumption.

Theorem 4 (Weak isomorphism preserves TINI). *If a language L is weakly isomorphic to a language L', and L' satisfies TINI, then L satisfies TINI.*

Proof. Shown by transporting configurations and reduction derivations from L to L', applying TINI and transporting the resulting equivalence back using functoriality of f^{-1} over l-equivalences. □

Unfortunately, an isomorphism is often too strong of a requirement. To obtain an isomorphism with our single heap semantics, we need to mimic the behavior of several heaps with a single actual heap. The interesting cases are when we sandbox an expression and when messages are sent and received. The rule for sandboxing is parametrized by the strategy κ (see Section 2), which defines what heap the new task should execute with. We have considered two choices:

- When we sandbox into an empty heap, existing addresses in the sandboxed expression are no longer valid and the task will get stuck (and then removed by I-noStep). Thus, we must rewrite the sandboxed expression so that all addresses point to fresh addresses guaranteed to not occur in the heap. Similarly, sending a memory address should be rewritten.
- When we clone the heap, we have to copy everything reachable from the sandboxed expression and replace all addresses correspondingly. Even worse, the behavior of sending a memory address now depends on whether that address existed at the time the receiving task was sandboxed; if it did, then the address should be rewritten to the existing one.

Isomorphism demands we implement this convoluted behavior, despite our initial motivation of a more efficient implementation.

4.1 Restricting the IFC Language

A better solution is to forbid sandboxed expressions as well as messages sent to other tasks to contain memory addresses in the first place. In a statically typed language, the type system could prevent this from happening. In dynamically typed languages such as λ_{ES}, we might restrict the transition for **sandbox** and **send** to only allow expressions without memory addresses.

While this sounds plausible, it is worth noting that we are modifying the IFC language semantics, which raises the question of whether non-interference is preserved. This question can be subtle: it is easy to remove a transition from a language and invalidate TSNI. Intuitively if the restriction depends on secret data, then a public thread can observe if some other task terminates or not, and from that obtain information about the secret data that was used to restrict the transition. With this in mind, we require semantic rules to get restricted only based on information observable by the task triggering them. This ensures that non-interference is preserved, as the restriction does not depend on confidential information. Below, we give the formal definition of this condition for the abstract IFC language $L_{\text{IFC}}(\alpha, \lambda)$.

Definition 6 (Restricted IFC language). *For a family of predicates $\mathcal{P}$ (one for every reduction rule), we call $L^{\mathcal{P}}_{IFC}(\alpha, \lambda)$ a restricted IFC language if its* *definition is equivalent to the abstract language $L_{IFC}(\alpha, \lambda)$, with the following exception: the reduction rules are restricted by adding a predicate $P \in \mathcal{P}$ to the premise of all rules other than* I-noStep*. Furthermore, the predicate P can depend only on the erased configuration $\varepsilon_l(c)$, where l is the label of the first task in the task list and c is the full configuration.*

By the following theorem, the restricted IFC language with an appropriate scheduling policy is non-interfering.

Theorem 5. *For any target language λ and family of predicates $\mathcal{P}$, the restricted IFC language $L^{\mathcal{P}}_{IFC}(\mathrm{RR}, \lambda)$ is TSNI. Furthermore, the IFC language $L^{\mathcal{P}}_{IFC}(\textsc{Seq}, \lambda)$ is TINI.*

In the extended version of this paper we give an example how this formalism can be used to show non-intereference of an implementation of IFC with a single heap.

5 Real World Languages

Our approach can be used to retrofit any language for which we can achieve isolation with information flow control. Unfortunately, controlling the external effects of a real-world language, as to achieve isolation, is language-specific and varies from one language to another.[6] Indeed, even for a single language (e.g., JavaScript), how one achieves isolation may vary according to the language runtime or embedding (e.g., server and browser).

In this section, we describe several implementations and their approaches to isolation. In particular, we describe two JavaScript IFC implementations building on the theoretical foundations of this work. Then, we consider how our formalism could be applied to the C programming language and connect it to a previous IFC system for Haskell.

5.1 JavaScript

JavaScript, as specified by ECMAScript [14], does not have any built-in functionality for I/O. For this language, which we denote by λ_{JS}, the IFC system $L_{\text{IFC}}(\text{RR}, \lambda_{\text{JS}})$ can be implemented by exposing IFC primitives to JavaScript as part of the runtime, and running multiple instances of the JavaScript virtual machine in separate OS-level threads. Unfortunately, this becomes very costly when a system, such as a server-side web application, relies on many tasks.

Luckily, this issue is not unique to our work—browser layout engines also rely on isolating code executing in separate iframes (e.g., according to the same-origin policy). Since creating an OS thread for each iframe is expensive, both the V8 and SpiderMonkey JavaScript engines provide means for running JavaScript code in isolation within a single OS thread, on disjoint sub-heaps. In V8, this unit of isolation is called a *context*; in SpiderMonkey, it is called a *compartment*. (We will use these terms interchangeably.) Each context is associated with a global object, which, by default, implements the JavaScript standard library

[6] Though we apply our framework to several real-world languages, it is conceivable that there are languages for which isolation cannot be easily achieved.

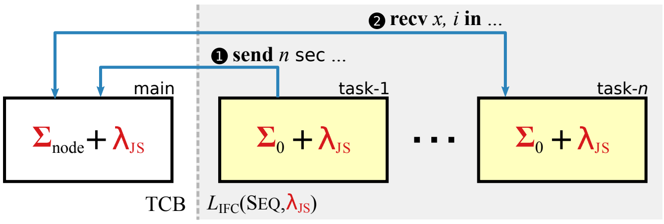

❷ recv x, i in ...
❶ send n sec ...
main
task-1
task-n
$\Sigma_{\text{node}} + \lambda_{\text{JS}}$
$\Sigma_0 + \lambda_{\text{JS}}$
...
$\Sigma_0 + \lambda_{\text{JS}}$
TCB
$L_{\text{IFC}}(\text{Seq}, \lambda_{\text{JS}})$

Fig. 6: This example shows how our trusted monitor (left) is used to mediate communication between two tasks for which IFC is enforced (right).

(e.g., `Object`, `Array`, etc.). Naturally, we adopt contexts to implement our notion of tasks.

When JavaScript is embedded in browser layout engines, or in server-side platforms such as Node.js, additional APIs such as the Document Object Model (DOM) or the file system get exposed as part of the runtime system. These features are exposed by extending the global object, just like the standard library. For this reason, it is easy to modify these systems to forbid external effects when implementing an IFC system, ensuring that important effects can be reintroduced in a safe manner.

Server-side IFC for Node.js: We have implemented $L_{\text{IFC}}(\textsc{Seq}, \lambda_{\text{JS}})$ for Node.js in the form of a library, without modifying Node.js or the V8 JavaScript engine. Our implementation[7] provides a library for creating new tasks, i.e., contexts whose global object only contains the standard JavaScript library and our IFC primitives (e.g., **send** and **sandbox**). When mapped to our formal treatment, **sandbox** is defined with $\kappa(\mathbf{\Sigma}) = \mathbf{\Sigma}_0$, where $\mathbf{\Sigma}_0$ is the global object corresponding to the standard JavaScript library and our IFC primitives. These IFC operations are mediated by the trusted library code (executing as the main Node.js context), which tracks the state (current label, messages, etc.) of each task. An example for **send**/**recv** is shown in Fig. 6. Our system conservatively restricts the kinds of messages that can be exchanged, via **send** (and **sandbox**), to string values. In our formalization, this amounts to restricting the IFC language rule for **send** in the following way:

$$\textsc{JS-send}\quad \frac{\begin{array}{ccc} l \sqsubseteq l' & \Sigma\,(i') = \Theta & \Sigma' = \Sigma\,[i' \mapsto (l', i, v), \Theta] \\ e = {}^{\text{IT}}\lceil \mathbf{e} \rceil & \multicolumn{2}{c}{\mathcal{E}_{\mathbf{\Sigma}}\,[\texttt{typeOf}(\mathbf{e})\ \texttt{===}\ \texttt{"string"}] \rightarrow \mathcal{E}_{\mathbf{\Sigma}}\,[\texttt{true}]} \end{array}}{\Sigma; \langle \mathbf{\Sigma}, E[\mathbf{send}\ i'\ l'\ v]_I \rangle_l^i, \ldots \hookrightarrow \Sigma'; \alpha_{\text{step}}(\langle \mathbf{\Sigma}, E[\langle\rangle]_I \rangle_l^i, \ldots)}$$

Of course, we provide a convenience library which marshals JSON objects to/from strings. We remark that this is not unlike existing message-passing JavaScript APIs, e.g., `postMessage`, which impose similar restrictions as to avoid sharing references between concurrent code.

While the described system implements $L_{\text{IFC}}(\textsc{Seq}, \lambda_{\text{JS}})$, applications typically require access to libraries (e.g., the file system library `fs`) that have external effects. Exposing the Node.js APIs directly to sandboxed tasks is unsafe. Instead,

[7] Available at `http://github.com/deian/espectro`.

we implement libraries (like a labeled version of `fs`) as message exchanges between the sandboxed tasks (e.g., `task-1` in Fig. 6) and the main Node.js task that implements the IFC monitor. While this is safer than simply wrapping unsafe objects, which can potentially be exploited to access objects outside the context (e.g., as seen with ADSafe [38]), adding features such as the `fs` requires the code in the main task to ensures that labels are properly propagated and enforced. Unfortunately, while imposing such a proof burden is undesirable, this also has to be expected: different language environments expose different libraries for handling external I/O, and the correct treatment of external effects is application specific. We do not extend our formalism to account for the particular interface to the file system, HTTP client, etc., as this is specific to the Node.js implementation and does not generalize to other systems.

Client-side IFC: This work provides the formal basis for the core part of the COWL client-side JavaScript IFC system [37]. Like our Node.js implementation, COWL takes a coarse-grained approach to providing IFC for JavaScript programs. However, COWL's IFC monitor is implemented in the browser layout engine instead (though still leaving the JavaScript engine unmodified).

Furthermore, COWL repurposes existing contexts (e.g., iframes and pages) as IFC tasks, only imposing additional constraints on how they communicate. As with Node.js, at its core, the global object of a COWL task should only contain the standard JavaScript libraries and `postMessage`, whose semantics are modeled by our JS-SEND rule. However, existing contexts have objects such as the DOM, which require COWL to restrict a task's external effects. To this end, COWL mediates any communication (even via the DOM) at the context boundary.

Simply disallowing all the external effects is overly-restricting for real-world applications (e.g., pages typically load images, perform network requests, etc.). In this light, COWL allows safe network communication by associating an implicit label with remote hosts (a host's label corresponds to its origin). In turn, when a task performs a request, COWL's IFC monitor ensures that the task label can flow to the remote origin label. While the external effects of COWL can be formally modeled, we do not model them in our formalism, since, like for the Node.js case, they are specific to this system.

5.2 Haskell

Our work borrows ideas from the LIO Haskell coarse-grained IFC system [33, 34]. LIO relies on Haskell's type system and monadic encoding of effects to achieve isolation and define the IFC sub-language. Specifically, LIO provides the `LIO` monad as a way of restricting (almost all) side-effects. In the context of our framework, LIO can be understood as follows: the *pure subset* of Haskell is the target language, while the monadic subset of Haskell, operating in the `LIO` monad, is the IFC language.

Unlike our proposal, LIO originally associated labels with exceptions, in a similar style to fine-grained systems [21, 35]. In addition to being overly complex, the interaction of exceptions with clearance (which sets an upper bound on the floating label, see the extended version of this paper) was incorrect: the clearance was restored to the clearance at point of the catch. Furthermore, pure exceptions (e.g., divide by zero) always percolated to trusted code, effectively allowing for denial of service attacks. The insights gained when viewing coarse-grained IFC as presented in this paper led to a much cleaner, simpler treatment of exceptions, which has now been adopted by LIO.

5.3 C

C programs are able to execute arbitrary (machine) code, access arbitrary memory, and perform arbitrary system calls. Thus, the confinement of C programs must be imposed by the underlying OS and hardware. For instance, our notion of isolation can be achieved using Dune's hardware protection mechanisms [5], similar to Wedge [5, 7], but using an information flow control policy. Using page tables, a (trusted) IFC runtime could ensure that each task, implemented as a lightweight process, can only access the memory it allocates—tasks do not have access to any shared memory. In addition, ring protection could be used to intercept system calls performed by a task and only permit those corresponding to our IFC language (such as **getLabel** or **send**). Dune's hardware protection mechanism would allow us to provide a concrete implementation that is efficient and relatively simple to reason about, but other sandboxing mechanisms could be used in place of Dune.

In this setting, the combined language of Section 2 can be interpreted in the following way: calling from the target language to the IFC language corresponds to invoking a system call. Creating a new task with the **sandbox** system call corresponds to *forking* a process. Using page tables, we can ensure that there will be no shared memory (effectively defining $\kappa(\boldsymbol{\Sigma}) = \boldsymbol{\Sigma}_0$, where $\boldsymbol{\Sigma}_0$ is the set of pages necessary to bootstrap a lightweight process). Similarly, control over page tables and protection bits allows us to define a **send** system call that copies pages to our (trusted) runtime queue; and, correspondingly, a **recv** that copies the pages from the runtime queue to the (untrusted) receiver. Since C is not memory safe, conditions on these system calls are meaningless. We leave the implementation of this IFC system for C as future work.

6 Extensions and Limitations

While the IFC language presented thus far provides the basic information flow primitives, actual IFC implementations may wish to extend the minimal system with more specialized constructs. For example, COWL provides a labeled version of the XMLHttpRequest (XHR) object, which is used to make network requests. Our system can be extended with constructs such as labeled values, labeled mutable references, clearance, and privileges. For space reasons, we provide details of this, including the soundness proof with the extensions, in the appendix of the extended version of this paper. Here, we instead discuss a limitation of our formalism: the lack of external effects.

Specifically, our embedding assumes that the target language does not have any primitives that can induce external effects. As discussed in Section 5, imposing this restriction can be challenging. Yet, external effects are crucial when implementing more complex real-world applications. For example, code in an IFC browser must load resources or perform XHR to be useful.

Like labeled references, features with external effects must be modeled in the IFC language; we must reason about the precise security implications of features that otherwise inherently leak data. Previous approaches have modeled external effects by internalizing the effects as operations on labeled channels/references [34]. Alternatively, it is possible to model such effects as messages to/from certain labeled tasks, an approach taken by our Node.js implementation. These "special" tasks are trusted with access to the unlabeled primitives that can be used to perform the external effects; since the interface to these tasks is already part of the IFC language, the proof only requires showing that this task does not leak information. Instead of restricting or wrapping unsafe primitives, COWL allow for controlled network communication at the context boundary. (By restricting the default XHR object, for example, COWL allows code to communicate with hosts according to the task's current label.)

7 Related Work

Our information flow control system is closely related to the coarse-grained information systems used in operating systems such as Asbestos [15], HiStar [45], and Flume [23], as well as language-based *floating-label IFC systems* such as LIO [33], and Breeze [21], where there is a monotonically increased label associated with threads of execution. Our treatment of termination-sensitive and termination-insensitive interference originates from Smith and Volpano [32, 40].

One information flow control technique designed to handle legacy code is secure multi-execution (SME) [13, 28]. SME runs multiple copies of the program, one per security level, where the semantics of I/O interactions is altered. Bielova et al. [6] use a transition system to describe SME, where the details of the underlying language are hidden. Zanarini et al. [44] propose a novel semantics for programs based on interaction trees [22], which treats programs as blackboxes about which nothing is known, except what can be inferred from their interaction with the environment. Similar to SME, our approach mediates I/O operations; however, our approach only runs the program once.

One of the primary motivations behind this paper is the application of information flow control to JavaScript. Previous systems retrofitted JavaScript with fine-grained IFC [18, 19]. While fine-grained IFC can result in fewer false alarms and target legacy code, it comes at the cost of complexity: the system must accommodate the entirety of JavaScript's semantics [19]. By contrast, coarse-grained approaches to security tend to have simpler implications [11, 43].

The constructs in our IFC language, as well as the behavior of inter-task communication, are reminiscent of distributed systems like Erlang [2]. In distributed systems, isolation is required due to physical constraints; in information flow control, isolation is required to enforce non-interference. Papagiannis et al. [27] built an information flow control system on top of Erlang that shares some similarities to ours. However, they do not take a floating-label approach (processes can find out when sending a message failed due to a forbidden information flow), nor do they provide security proofs.

There is limited work on general techniques for retrofitting arbitrary languages with information flow control. However, one time-honored technique is to define a fundamental calculus for which other languages can be desugared into. Abadi et al. [1] motivate their core calculus of dependency by showing how various previous systems can be encoded in it. Tse and Zdancewic [39], in turn, show how this calculus can be encoded in System F via parametricity. Broberg and Sands [9] encode several IFC systems into Paralocks. However, this line of work is primarily focused on static enforcements.

8 Conclusion

In this paper, we argued that when designing a coarse-grained IFC system, it is better to start with a fully isolated, multi-task system and work one's way back to the model of a single language equipped with IFC. We showed how systems designed this way can be proved non-interferent without needing to rely on details of the target language, and we provided conditions on how to securely refine our formal semantics to consider optimizations required in practice. We connected our semantics to two IFC implementations for JavaScript based on this formalism, explained how our methodology improved an exiting IFC system for Haskell, and proposed an IFC system for C using hardware isolation. By systematically applying ideas from IFC in operating systems to programming languages for which isolation can be achieved, we hope to have elucidated some of the core design principles of coarse-grained, dynamic IFC systems.

Acknowledgements We thank the POST 2015 anonymous reviewers, Adriaan Larmuseau, Sergio Maffeis, and David Mazières for useful comments and suggestions. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, by the NSF, by the AFOSR, by multiple gifts from Google, by a gift from Mozilla, and by the Swedish research agencies VR and the Barbro Oshers Pro Suecia Foundation. Deian Stefan and Edward Z. Yang were supported by the DoD through the NDSEG.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A Core Calculus of Dependency. In *POPL*, 1999.

[2] J. Armstrong. Making reliable distributed systems in the presence of software errors. 2003.

[3] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. ESORICS, 2008.

[4] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *ESORICS*, 2007.

[5] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *OSDI*, 2012.

[6] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *NSS*, 2011.

[7] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. In *NSDI*, 2008.

[8] Boudol and Castellani. Noninterference for concurrent programs. In *ICALP*, 2001.

[9] N. Broberg and D. Sands. Paralocks: Role-based information flow control and beyond. In *POPL*, 2010.

[10] P. Buiras, A. Levy, D. Stefan, A. Russo, and D. Mazières. A library for removing cache-based attacks in concurrent information flow systems. In *TGC*, 2013.

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

[12] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5), 1976.

[13] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *SP*, 2010.

[14] Ecma International. ECMAScript language specification. http://www.ecma.org/, 2014.

[15] 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 *SOSP*, 2005.

[16] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *TCS*, 103(2), 1992.

[17] J. Goguen and J. Meseguer. Security policies and security Models. In *SP*, 1982.

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

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

[20] S. Heule, D. Stefan, E. Z. Yang, J. C. Mitchell, and A. Russo. Ifc inside: Retrofitting languages with dynamic information flow control. http://cowl.ws/ifc-inside.pdf, 2015.

[21] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. In *SP*, 2013.

[22] B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS*, 62, 1997.

[23] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[24] P. Li and S. Zdancewic. Arrows for secure information flow. *TCS*, 411(19), 2010.

[25] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *POPL*, 2007.

[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, 2001.

[27] I. Papagiannis, M. Migliavacca, D. M. Eyers, B. Sh, J. Bacon, and P. Pietzuch. Enforcing user privacy in web applications using Erlang. In *W2SP*, 2010.

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

[29] A. Russo and A. Sabelfeld. Securing Interaction between threads and the scheduler. In *CSFW*, 2006.

[30] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Haskell*, 2008.

[31] V. Simonet. The Flow Caml system. Software release at http://cristal.inria.fr/~simonet/soft/flowcaml/, 2003.

[32] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL*, 1998.

[33] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell*, 2011.

[34] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *ICFP*, 2012.

[35] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457*, 2012.

[36] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *ESORICS*, 2013.

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

[38] A. Taly, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical javascript apis. In *SP*, 2011.

[39] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *ICFP*, 2004.

[40] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *CSFW*, 1997.

[41] W3C. HTML5 web messaging. http://www.w3.org/TR/webmessaging/, 2012.

[42] E. Z. Yang and D. Mazières. Dynamic space limits for Haskell. In *PLDI*, 2014.

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

[44] D. Zanarini, M. Jaskelioff, and A. Russo. Precise enforcement of confidentiality for reactive systems. In *CSF*, 2013.

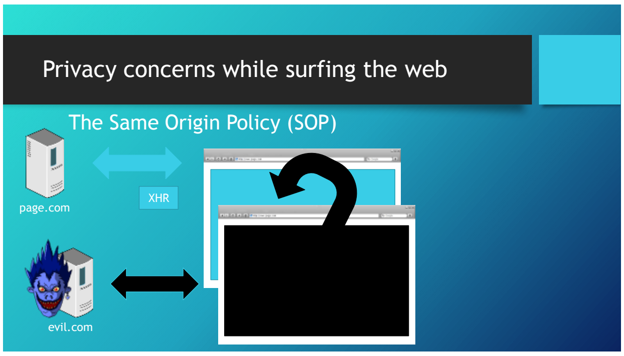
[45] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

Bonus: The Most Dangerous Code in Your Browser

Security in Browser Extensions

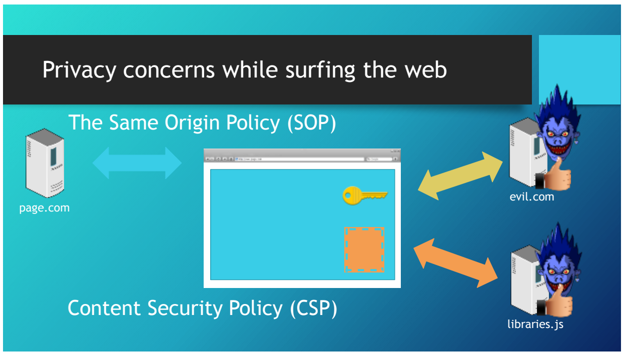
[HotOS 2015]

Alejandro Russo
russo@chalmers.se



Privacy concerns while surfing the web

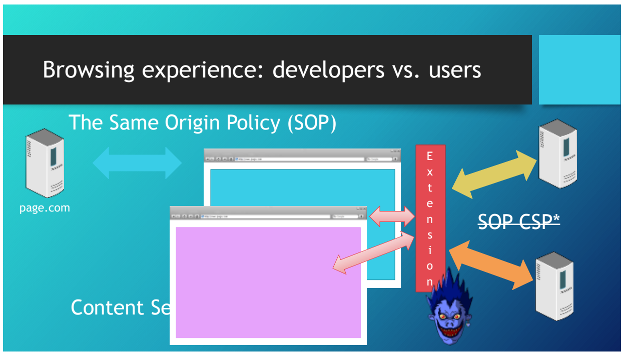
The Same Origin Policy (SOP)



Privacy concerns while surfing the web

The Same Origin Policy (SOP)

Content Security Policy (CSP)



Browsing experience: developers vs. users

The Same Origin Policy (SOP)

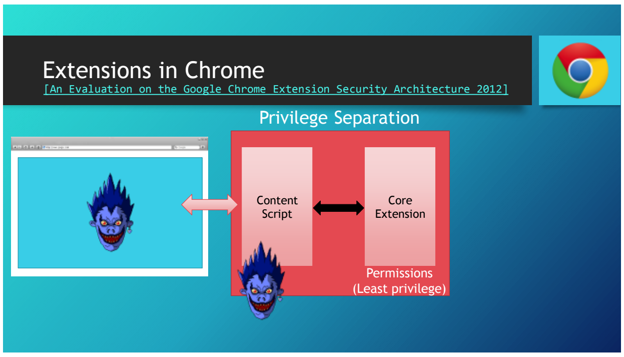
Content Se

Add-ons in Firefox

[Abusing Exploiting and Pwning with Firefox Add-ons 2013]

"The add-on code is fully trusted by Firefox. The installation of malicious add-ons can result in full system compromise."

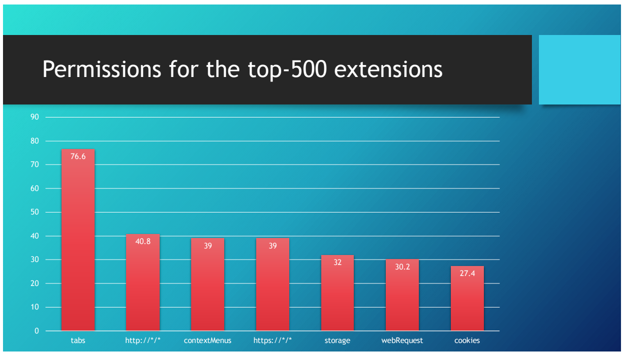
KEEP
CALM
AND
INSTALL
EXTENSIONS



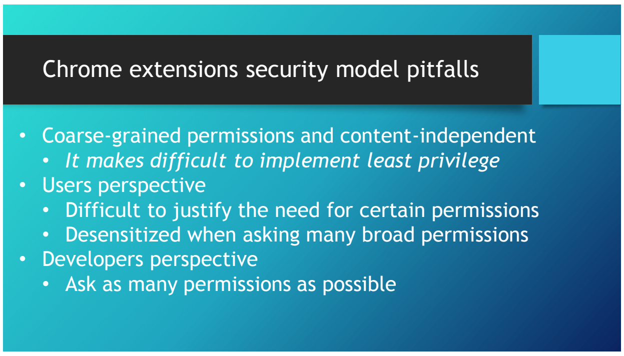
Extensions in Chrome

[An Evaluation on the Google Chrome Extension Security Architecture 2012]

Privilege Separation

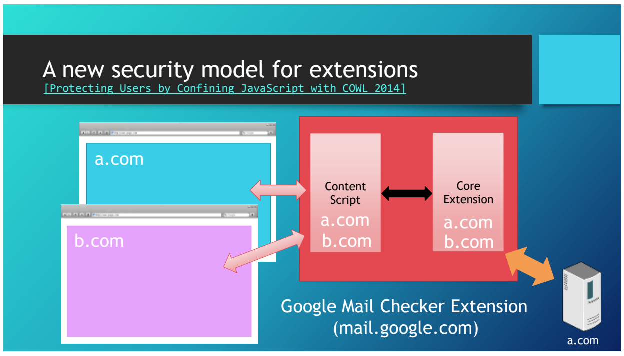


Permissions for the top-500 extensions



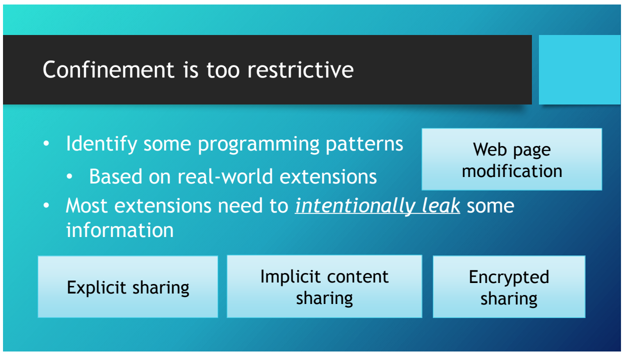
Chrome extensions security model pitfalls

- Coarse-grained permissions and content-independent
  - *It makes difficult to implement least privilege*
- Users perspective
  - Difficult to justify the need for certain permissions
  - Desensitized when asking many broad permissions
- Developers perspective
  - Ask as many permissions as possible



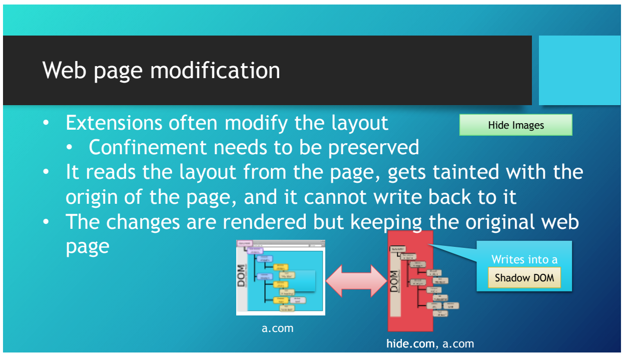
A new security model for extensions

[Protecting Users by Confining JavaScript with COWL 2014]



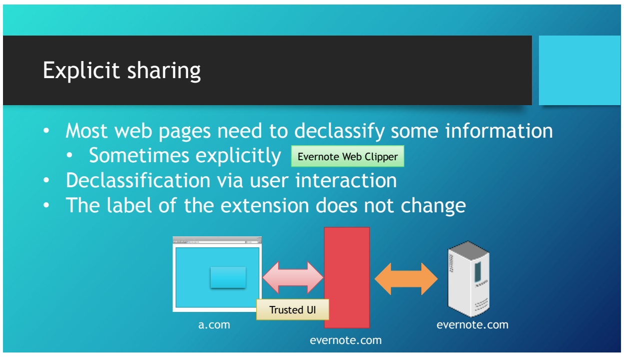
Confinement is too restrictive

- Identify some programming patterns
  - Based on real-world extensions
- Most extensions need to *intentionally leak* some information



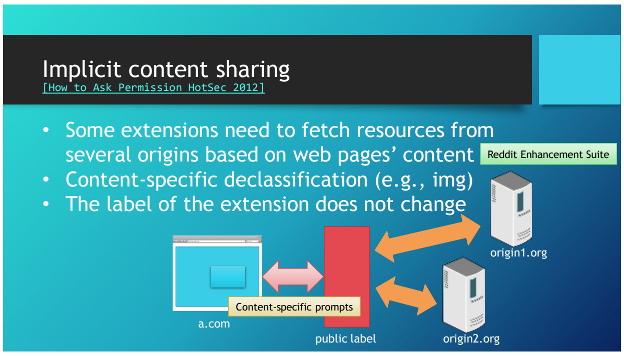
Web page modification

- Extensions often modify the layout
  - Confinement needs to be preserved
- It reads the layout from the page, gets tainted with the origin of the page, and it cannot write back to it
- The changes are rendered but keeping the original web page



Explicit sharing

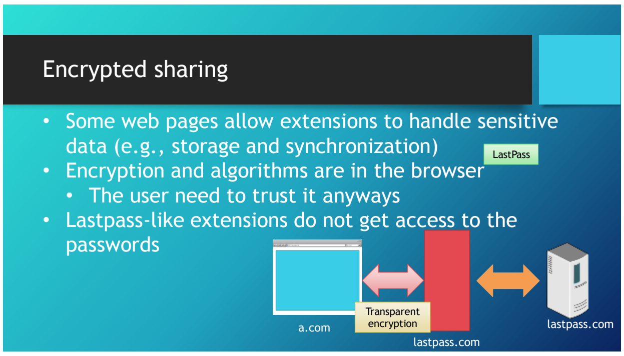
- Most web pages need to declassify some information
  - Sometimes explicitly
- Declassification via user interaction
- The label of the extension does not change



Implicit content sharing

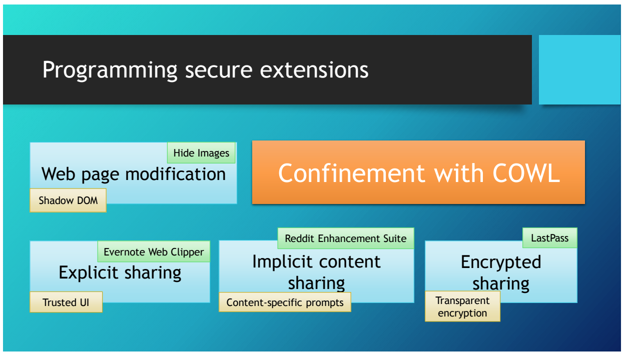
[How to Ask Permission HotSec 2012]

- Some extensions need to fetch resources from several origins based on web pages' content
- Content-specific declassification (e.g., img)
- The label of the extension does not change



Encrypted sharing

- Some web pages allow extensions to handle sensitive data (e.g., storage and synchronization)
- Encryption and algorithms are in the browser
  - The user need to trust it anyways
- Lastpass-like extensions do not get access to the passwords



Programming secure extensions

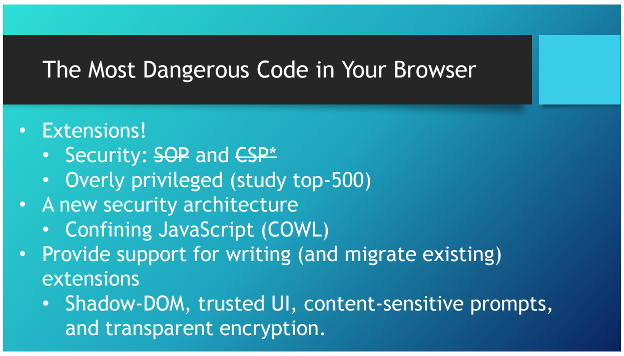
Hide Images
Web page modification
Shadow DOM

Confinement with COWL

Evernote Web Clipper
Explicit sharing
Trusted UI

Reddit Enhancement Suite
Implicit content sharing
Content-specific prompts

LastPass
Encrypted sharing
Transparent encryption



The Most Dangerous Code in Your Browser

- Extensions!
  - Security: ~~SOP~~ and ~~CSP*~~
  - Overly privileged (study top-500)
- A new security architecture
  - Confining JavaScript (COWL)
- Provide support for writing (and migrate existing) extensions
  - Shadow-DOM, trusted UI, content-sensitive prompts, and transparent encryption.

The Most Dangerous Code in the Browser

Stefan Heule[1] Devon Rifkin[1] Alejandro Russo*[2] Deian Stefan[1]
[1]Stanford University [2]Chalmers University of Technology

ABSTRACT

Browser extensions are ubiquitous. Yet, in today's browsers, extensions are the most dangerous code to user privacy. Extensions are third-party code, like web applications, but run with elevated privileges. Even worse, existing browser extension systems give users a false sense of security by considering extensions to be more trustworthy than web applications. This is because the user typically has to explicitly grant the extension a series of permissions it requests, e.g., to access the current tab or a particular website. Unfortunately, extensions developers do not request minimum privileges and users have become desensitized to install-time warnings. Furthermore, permissions offered by popular browsers are very broad and vague. For example, over 71% of the top-500 Chrome extensions can trivially leak the user's data from *any* site. In this paper, we argue for new extension system design, based on *mandatory access control*, that protects the user's privacy from malicious extensions. A system employing this design can enable a range of common extensions to be considered *safe*, i.e., they do not require user permissions and can be ensured to not leak information, while allowing the user to share information when desired. Importantly, such a design can make permission requests a rarity and thus more meaningful.

1 INTRODUCTION

The modern web browser is one of the most popular application platforms. This is, in part, because building and deploying web applications is remarkably easy and, in other part, because using such applications is even easier: a user simply needs to type in a URL to run sophisticated applications, such as document editors, email clients, or video players. Unlike venerable desktop applications, these *apps* run on many different devices without imposing painstaking installation procedures or forcing users to be concerned with security—e.g., the weather app stealing their banking data or locally-stored photos.

As the web evolved to address different application demands, it did so in a somewhat security-concious fashion. In particular, when adding a new feature (e.g., offline caching [22]), web browsers have been careful to ensure that the feature was confined to the browser, i.e., it did not unsafely expose underlying OS resources, and that it could not be used to violate the *same-origin policy (SOP)* [5, 29]. The SOP roughly dictates that an app from one origin can only read and write content from the same origin. This ensures that one app cannot interfere with another—it is the reason the weather app cannot read data from the tab running the banking app.

Unfortunately, the web platform has some natural limitations. Despite prioritizing "users over [app] authors [27]," a user's experience on the web is largely dictated by the app author. For example, the web does not provide users with a means for removing advertisements served by an app. Similarly, the user cannot directly share content from one app with another app of their choosing without the app author offering such a service. Of course, it is unrealistic to demand that app authors provide such features since they may be at odds with the authors' goals (e.g., to serve ads).

To address the limitations of the web platform, most modern browsers provide users with *extensions*. Extensions are typically used to modify and extend web application behavior, content, and display (style). For example, Adblock Plus [1], one of the most-widely used extensions, modifies apps by blocking network requests and hiding different page elements to provide ad-free browsing. However, extensions can also be used to implement completely new functionality. For instance, LastPass [2] allows users to store and retrieve credentials for arbitrary apps, in the cloud. And, in some cases, extensions even modify and extend the browser itself.

Unlike web applications, which are bound by the SOP, extensions can access the page contents of different-origin apps, perform arbitrary network communication, inspect and modify browser history, etc. Misusing such privileged APIs is a serious security concern. In light of this, browsers vendors have imposed various restrictions. For example, Chrome—which has the most comprehensive extension security system—makes it difficult to install extensions that are not distributed through its official Chrome Web Store (CWS), requires users to grant extensions access to use privileged APIs, and employs various mechanisms to prevent privilege-escalation attacks [6, 8, 21].

Unfortunately, even Chrome's extension system has fundamental shortcomings. For example, Chrome's attacker model assumes that extensions are not malicious, but rather that they are *benign-but-buggy* [6]. As a consequence, Chrome's security mechanisms were designed to prevent attacks wherein malicious app pages try to exploit vulnerable extensions. However, the system does

*Work conducted while at Stanford University.

not provide a way for protecting sensitive app data from extensions—a malicious extension can easily leak data. And the premise for placing more trust on extension code over web app code is unfounded: both are provided by third-party developers, while the former runs with elevated privileges. Unlike other privileged code in the browser (e.g., plugins), these JavaScript-based extensions are made available to users without a code review process. It is of no surprise that roughly 5% of the users visiting Google have at least one malicious extension installed, as showed by a recent study [7, 26].

Unfortunately, in the current extensions system, even trustworthy but vulnerable extensions can be exploited by malicious pages to leak sensitive data from cross-origin apps [20]. While Chrome's mechanisms limit an attacker to abusing the privileges held by the vulnerable extension, developer incentives have led many extensions request broad privileges. Similarly, many users have become desensitized to the install-time warning accompanying these extensions [12]. For example, of the 500 most popular Chrome extensions, over 71% request the privilege to "read and change all your data on the websites you visit." Since these extensions also retain their privileges throughout lifetime of the extension, this makes them especially attractive targets to attackers that wish to steal user-sensitive data.

In today's browsers, extensions are arguably the most dangerous code to user privacy. Yet, this need not be the case. This position paper argues for new extension system designs that can address user privacy without giving up on desired extension functionality.

What might such an extension system look like? Given that apps handle sensitive data such as banking information and that extensions are written by potentially untrusted third-party developers, it is clear that apps need to be protected from extensions. At the same time, it is important to keep protecting extensions from apps, as extensions may run with higher privileges. Mandatory access control (MAC)-based confinement [19, 25] naturally fits this scenario of mutually distrusting parties, where apps and extensions can be protected from one another.

But, MAC alone is not enough. While MAC-based confinement can prevent an extension from leaking sensitive app data even after it has access to it, for many extensions, this is overly restricting. For example, the Google Dictionary extension [3] needs to read text from the page and communicate with the network when looking up a word—its functionality relies on the ability to "leak" data. Hence, the extension system should allow users to explicitly share app data with extensions, which may further share the data with a remote server. Similarly, it should provide robust APIs that common extensions can use to operate on sensitive information without being confined. Together with MAC-based confinement this can alleviate the need for permissions altogether for a broad range of *safe* extensions: many extensions only read sensitive data and provide useful features to the user, but never disseminate the data without user intent.

Of course, leveraging user actions to share data is not possible in all cases; user-approved permission may still be necessary. However, these permissions should be fine-grained and content-specific. Since many extensions are safe and do not rely on special permissions, it would be possible for the extension system to give users more meaningful messages and warn them appropriately about installing dangerous extensions.

In the rest of the paper we give a brief overview of Chrome's extension system and its limitations (§2). We then expand on the design goals of an extensions system that addresses Chrome's limitations (§3) and describe a preliminary system design that satisfies these goals (§4). Finally, we conclude (§5).

2 Chrome's Extension Model

In this paper, we focus on the Chrome extension model, whose security system is widely regarded as being more advanced than those implemented in other browsers [8, 16]. More specifically, we focus on JavaScript-based extensions; we do not consider plugins, which can additionally execute native code.[1] Below we describe the extension system's security model, evaluate the use of permissions in this ecosystem, and highlight its key limitations.

2.1 Security Model

The Chrome attacker model assumes that extensions are trustworthy, but vulnerable to attacks carried out by apps [6]. Hence, Chrome's extension security system is designed to protect extensions from apps. Chrome requires developers to *privilege-separate* [21] extensions into a *content script* and a *core extension*. Content scripts interact directly with web pages (e.g., by reading the page's cookies or modifying its DOM),[2] but do not have access to any privileged APIs. To perform privileged operations, content scripts use message-passing to communicate with core extension scripts, which have access to the privileged APIs needed to perform the actions.

To mitigate the impact of exploits that compromise vulnerable content scripts, in addition to privilege separation, Chrome also follows the *principle of least privilege* [23]. Specifically, Chrome implements a permission system that can be used to limit the privileges available to core script extensions. By limiting the privileges of an

[1] Plugins make up only a small fraction of the space, require a code review before being put on the CWS, and are widely-accepted to be dangerous. We do not discuss them further.

[2] Actually, Chrome employs *isolated worlds* [6] to separate the JavaScript heaps of the content script and page. This prevents attacks where a malicious page redefines functions (e.g., `getElementById`) that are commonly used by extensions.

extension, the damages that can be caused from exploits is also more limited.

To this end, Chrome requires extension authors to statically declare, in a *manifest*, what kind of permissions the extension requires. In turn, the user must approve these permissions when installing the extension. Since the compromise of an overly-privileged extension can cause serious harm (e.g., leaking user's banking information), Chrome encourages developers to only request minimal privileges. Below, we report the results of our study evaluating permission usage in Chrome extensions.

2.2 Permission Study

We surveyed the permissions used by the 500 most popular Chrome extensions [14] by inspecting their manifest files.[3] Most extensions are widely deployed: the most popular extension is used by more than 10 million users; the 500th extension is used by more than 76,000 users. In Table 1, we list the permissions most often requested. The most widely required permission is `tabs`, which among other abilities, allows an extension to retrieve URLs as they are navigated to. More concerning is the prevalence of permissions such as `http://*/*`, `https://*/*` and `<all_urls>`, which allow an extension to make requests to any origin (over HTTP, HTTPS, or both, respectively). Upon installing any extension that requires one of these permissions (or several other similarly high-privilege permissions), the user is warned that the extension can "read and change all [their] data on the websites [they] visit." These permissions can easily be used maliciously, for example, to retrieve a sensitive webpage (using the cookies stored in the browser) and forward its contents to the attacker's own server. Despite this danger, permissions triggering this warning are widely used. In our study, we found more than 71% of the top 500 extensions display this "read and change..." warning at installation-time. For users installing popular Chrome extensions, the norm is to allow for such high privilege requests. In fact, the more popular extensions are more likely to show this warning: 74% of the top 250 extensions display this warning, 82% of the top 100, and 88% of the top 50. We did not investigate how many of these extensions actually needed or exercised their requested permissions.

2.3 Pitfalls

Chrome assumes extensions to be benign-but-buggy [6]. Unfortunately, this trust in extensions is amiss, as extensions are written by potentially untrusted developers. For example, Kapravelos et al. [18] report on 140 malicious extension in the CWS. While taking malicious extensions out of the CWS is an appropriate response, this weak-attacker model has unfortunately led to the design of security mechanisms that do not explicitly protect web app data. This is particularly disconcerting because, as our study shows, most extensions can access highly sensitive data and communicate with the network—vulnerabilities in such extensions can be used to leak user data [9, 20]. A single compromised or malicious extension is enough to put the users privacy at risk.

Permission	Count	Permission	Count
`tabs`	*75.6%*	`webRequestBlocking`	*25.6%*
`storage`	*38.4%*	`cookies`	*24.6%*
`http://*/*`	*37.8%*	`unlimitedStorage`	*20.4%*
`https://*/*`	*36.4%*	`<all_urls>`	*19.2%*
`contextMenus`	*36.0%*	`webNavigation`	*16.6%*
`webRequest`	*32.2%*	`management`	*14.6%*
`notifications`	*30.4%*	`history`	*10.4%*

Table 1: The 14 most prevalent permissions as required by the top 500 Chrome extensions. A single extension may request any number of permissions. A full list explaining what each permission grants is available in [13].

Chrome provides a permission system that is meant to implement least privilege. Unfortunately, the explanations accompanying the permissions are broad and content-independent. Moreover, they do not convey to the user why such permissions are justified. Permissions must be accepted at install time, before a user has acquired context from using the extension.[4] Because the vast majority of extensions require many broad permissions, users have grown desensitized and accustomed to accepting most permission requests.

The other pitfall of the extension system is that it makes it difficult even for security conscious extension developers to request minimal privileges. The permissions are coarse grained and Chrome does not provide a way for requesting finer-grained access. Even worse, developers are incentivized to ask for more permission than they actually need. For example, if an extension update requires additional permissions (e.g., because of a new feature in the extension), Chrome automatically disables the extension until the user approves the new permissions. Since users get irritated by such prompts, developers often ask for more permissions than necessary up front thereby eliminating the risk of removal.

3 Design Goals

In this section, we outline a series of design goals that a modern browser extension system should strive for in order to protect user privacy and avoid Chrome's pitfalls.

1. **Handle mutually distrusting code** Extensions and web apps may be written by mutually distrusting parties. In addition to protecting extensions from untrusted app

[3] The manifests in this study were fetched on April 20, 2015.

[4] Chrome more recently added *optional permissions*, which, while still declared statically, only demand the user's approval at run-time, e.g., right before the extension uses the privileged API. Unfortunately optional permission warnings fall victim to the coarseness of the system and often ask for far more expansive abilities than required.

code, an extension system should provide mechanisms for protecting sensitive user (app) data from untrusted extensions without giving up on functionality. We assume a relatively strong attacker model where an extension executes attacker-provided code in attempt to leak user data via the extension system APIs. We consider leaks via covert channels to be out of scope.

2. **Leverage user intent** An extensions system should leverage user intent for security decisions. The system should provide APIs and trusted UIs for making security decisions part of the user's work-flow. For example, browsers can use user intent to make sharing of app data with an extension explicit via a sharing-menu API. A challenge with this goal is designing APIs and UIs that are not susceptible to *confused deputy attacks* [17].

3. **Provide a meaningful permission system** Most common extensions should not need to request user permission to perform their tasks. In the rare case that an extension requires to leak sensitive data without explicit user intent, the permissions available should be fine-grained and content-specific. Furthermore, the system should provide the user with specific-enough information necessary to make an educated decision. This could happen, for instance, by asking for permission at runtime when the leaked content can be shown and the user has an idea of what the extension is about to do (in contrast to install-time permissions).

4. **Incentivize safety** The incentives of developers and the security model should align such that most common extensions are safe, i.e., they run without requiring user approval for permissions. The extension system should reward developers that implement these and other least-privileged extensions and penalize overly-privileged ones. For example, extensions that require permissions should require a security audit before being allowed to be installed. This ensures that APIs that leverage user intent for disclosing data are prioritized and that security warnings remain meaningful.

4 Preliminary Design

In this section we propose a new extension system designed to meet the aforementioned goals. We observe that, for an extension to be useful, it typically needs to have access to sensitive data such as the current app's URL or different parts of the page. However, if this information cannot be arbitrarily disseminated (within or external to the confines of the browser), then the user's privacy is not at risk: it is entirely safe for an extension to read sensitive data as long as it does not write it to an end-point that is not permitted by the SOP.

This idea of allowing code to compute on sensitive data, but restrict where it can subsequently write it, is endemic to MAC-based confinement systems (e.g., HiStar [28] and COWL [25]). In such systems, the sensitivity of information is tracked throughout the system and the security mechanism ensures that leaks due to data- and control-flow cannot occur.

We propose to extend the Chrome architecture to use a coarse-grained confinement system, similar to COWL [25]. As in Chrome, to achieve isolation and protect an extension from an untrusted app, every app and extension runs in a separate execution context. However, and unlike Chrome, our proposed extension system additionally protects app user data from an untrusted extension by ensuring that whenever the extension accesses sensitive data, its context gets "tainted" with the app's origin—we consider any data in the page to be sensitive. In turn, the origins with which the extension can subsequently communicate with is restricted by this taint—e.g., the extension cannot perform arbitrary network requests once it has read sensitive data.

With confinement, extensions that only read sensitive information can be implemented securely and without requiring any permissions. For instance, consider the Chrome extension Google Mail Checker [15], which displays an icon in the browser with the number of unread emails in Gmail. Confinement allows this extensions to connect to Gmail using the users credentials. Once it does so, however, the execution context is tainted with `mail.google.com` and thus cannot communicate with, for instance, `evil.com`. However, the extension can safely do its job and show an unread count to the user. We remark that such a system satisfies our goal of protecting user data against malicious extensions—even if malicious, the extension cannot leak the user's emails.

Of course, not all extensions are this simple, and a real extension system must provide extension developers with APIs to carry out common tasks. Below we describe some of these APIs, and in particular focus on APIs that make the confinement system more flexible or address our design goals directly.

Page access Some extensions read and modify page contents. Our system provides content script extensions with APIs for reading and writing the DOM of a page, much like COWL's labeled DOM workers [25]. Importantly, when accessing the DOM of a page, the content script is tainted with the origin of the page and its functionality is subsequently restricted to ensure that the read information is not leaked. (Of course, an extension can create such labeled content scripts at run time, to avoid over-tainting [25].) To ensure that extensions cannot leak through the page's DOM, we argue that extensions should instead write to a *shadow-copy* of the page DOM—any content loading as a result of modifying the shadow runs with the privilege of the extension and not the page. This ensures that the extension's changes to the page are isolated from that of the page, while giving the appearance of a single layout.

Explicit sharing Of course, some functionality requires sensitive data to be "leaked." For instance, Evernote Web Clipper [10] offers the functionality to save part of the page (e.g., the current selection) to `evernote.com`. Sharing page contents with arbitrary origins violates confinement—the page may contain sensitive information (e.g., a bank statement).

However, in some cases, the user may wish specific information to be sent to `evernote.com`, e.g., to save a recipe the user saw online. Confinement systems typically require *declassification* to allow such controlled leaks. However, extensions cannot be trusted to declassify data on their own. Our key insight is that information sharing typically follows a user action (e.g., clicking a "Save to Evernote" button), and therefore the intent of the user can be used to declassify the data. Concretely, we propose a sharing API that extensions can use to received data from the user and trusted browser UIs that users can employ to share specific content with these extensions, e.g., by means of a "Share with..." context menu entry. With this API, extensions like Evernote or Google Dictionary can be implemented without requiring specific declassification permissions. Of course, they can only leak data the user shares explicitly.

Encrypted sharing Since credentials are usually treated with more care than other data, our sharing API does not allow extensions to receive credentials without restrictions. Concretely, when sharing credentials, our sharing API provides extensions with *labeled blobs* [25], which the extension can only observe by tainting its context. However, it is often useful to allow extensions to synchronize and store such sensitive data. For this, we propose an API that takes a blob labeled with `a.com` and returns an unlabeled encrypted blob. This directly allows the extension to send the encrypted data to the cloud and synchronize it to another devices. There, a similar extension can use our API to decrypt the data to `a.com`-labeled credentials, which can then be used to, for example, fill in a `a.com` login form. This API makes our MAC system more flexible and directly allows the implementation of an extension to manage user passwords similar to LastPass. Unlike LasPass, however, the encryption algorithms and parameters are provided by the browser, only relying on the user to supply a master key.

Privileged content sharing Some extensions need to read content from the page and communicate with the network without user interaction. For example, the Reddit Enhancement Suite (RES) [4] fetches images that are linked in a post as to display them inline. Unfortunately, the page access API is insufficient when implementing such extensions since the code cannot communicate with arbitrary domains, as to fetch images, once it traverses the DOM to find the image links. Instead, we provide APIs that can be used to retrieve content from the page without imposing confinement restrictions. In particular, extension developers can request to access different kinds of elements on the page, e.g., URLs, or the current origin, etc. Our extension system would, in turn, ask the user to consent to the request at run time when the extension requests the data, applying the lessons and techniques of [11] to avoid desensitization (e.g., use different icons and colors to signify the severity of the request). Unlike Chrome's permissions requests, we envision providing users with content-specific choices (e.g., "RES wishes to see *all* the *links* on this page."), which they can also deny while continuing to use the extension—extensions should gracefully handle exceptions from these APIs or risk removal from the platform. Besides content-specific messages, other HCI techniques would be employed to make permissions more meaningful and to refrain users from blindly consenting to security prompts [24].

We remark that other APIs (e.g., a network API or declarative CSS replacement API) share many similarities with the above: they are fine-grained, content-driven, and abide by MAC. More interestingly, we note that our MAC-based approach encourages safe extensions, i.e., extensions that do not rely on privileges and raise alarms, but, rather, rely on sharing menu- and crypto-APIs to get user data. But in cases where permissions are required, our system presents security decisions at run-time and in terms of data—by reasoning about the content being disclosed users can make more educated decisions.

5 Summary

We identify extensions as some of the most dangerous code in the browser and show the pitfalls of modern extension security systems. For this reason, new extension security models that protect user privacy are in need. We outlined the goals of such as system and proposed a preliminary system design to this end. Our proposal relies on MAC-based confinement to prevent sensitive information from being arbitrarily shared. We also outlined several APIs that can be used to safely share data and make such a system flexible enough to handle a large class of common extensions, while keeping developer incentives aligned with security. We hope this encourages browser vendors to rethink extensibility.

Acknowledgements

We thank James Mickens, Petr Marchenko, Adrienne Porter Felt, and the anonymous reviewers for their helpful comments. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, by multiple gifts from Google, by a gift from Mozilla, by the Swedish research agency VR and the Barbro Oshers Pro Suecia Foundation.

References

[1] Adblock Plus – surf the web without annoying ads! https://adblockplus.org/, 2012. Visited April 21, 2015.

[2] LastPass password manager. https://lastpass.com/, 2012. Visited April 21, 2015.

[3] Google dictionary. https://chrome.google.com/webstore/detail/google-dictionary-by-goog/mgijmajocgfcbeboacabfgobmjgjcoja, 2015. Visited April 21, 2015.

[4] Reddit enhancement suite. http://redditenhancementsuite.com/, 2015. Visited April 21, 2015.

[5] Adam Barth. The web origin concept. https://tools.ietf.org/html/rfc6454, 2011. Visited April 21, 2015.

[6] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *NDSS*, 2010.

[7] BBC. Google purges bad extensions from Chrome. http://www.bbc.com/news/technology-32206511, 2015. Visited April 21, 2015.

[8] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. In *Security*. USENIX, 2012.

[9] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *ACSAC*, 2009.

[10] Evernote. Evernote web clipper. https://chrome.google.com/webstore/detail/evernote-web-clipper/pioclpoplcdbaefihamjohnefbikjilc, 2015. Visited April 21, 2015.

[11] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, David Wagner, et al. How to ask for permission. In *HotSec*. USENIX, 2012.

[12] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *WebApps'11*. USENIX, 2011.

[13] Google. Declare permissions. https://developer.chrome.com/extensions/declare_permissions, 2014. Visited April 21, 2015.

[14] Google. Chrome Web Store - Extensions. https://chrome.google.com/webstore/category/extensions?_sort=1, 2015. Visited April 21, 2015.

[15] Google. Google mail checker. https://chrome.google.com/webstore/detail/google-mail-checker/mihcahmgecmbnbcchbopgniflfhgnkff, 2015. Visited April 21, 2015.

[16] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *Security and Privacy*. IEEE, 2011.

[17] Norm Hardy. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS OS Review*, 22(4):36–38, 1988.

[18] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *Security*. USENIX, 2014.

[19] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[20] Petr Marchenko, Ulfar Erlingsson, and Brad Karp. Keeping sensitive data in browsers safe with ScriptPolice. Technical Report RN/13/02, UCL, January 2013.

[21] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Security*. USENIX, 2003.

[22] Alex Russell and Jungkee Song. Service workers. http://www.w3.org/TR/service-workers/, 2014. Visited April 21, 2015.

[23] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *IEEE*, 63(9), 1975.

[24] S. W. Smith. Humans in the loop: Human-computer interaction and security. *IEEE Security and Privacy*, 1(3), May 2003.

[25] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining JavaScript with COWL. In *OSDI*. USENIX, 2014.

[26] Kurt Thomas, Elie Bursztein, Chris Grierand Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon McCoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *Security and Privacy*. IEEE, 2015. To appear.

[27] Anne van Kesteren and Maciej Stachowiak. HTML design principles. `http://www.w3.org/TR/html-design-principles`, 2007. Visited April 21, 2015.

[28] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *OSDI*. USENIX, 2006.

[29] Michal Zelwski. Browser security handbook, part 2. `http://code.google.com/p/browsersec/wiki/Part2`, 2009. Visited April 21, 2015.

Contents