

A Model for Delimited Information Release

Andrei Sabelfeld
Chalmers University

Andrew C. Myers
Cornell University

presented by

Eric Hennigan
UC Irvine

Secure Software

Programs manipulate sensitive data

- Health Records
- Banking Accounts
- Identification Credentials

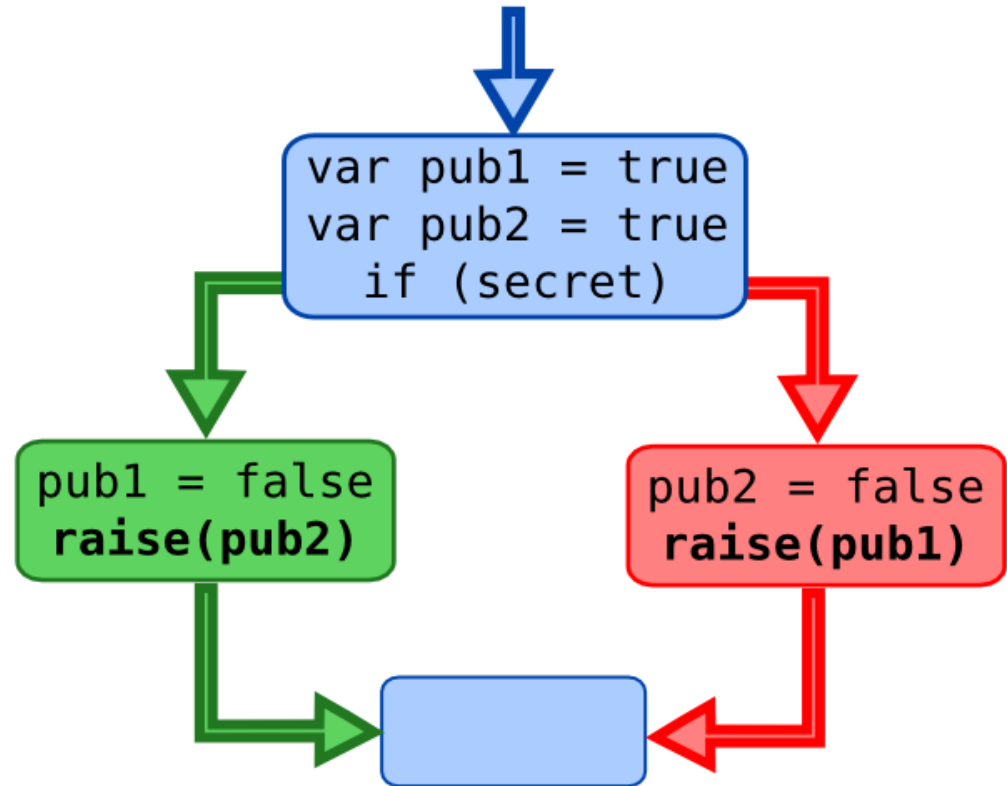
This data needs to be protected in some way



Image from ocw.mit.edu

Security-Typed Languages

Data Tainting
Information Flow



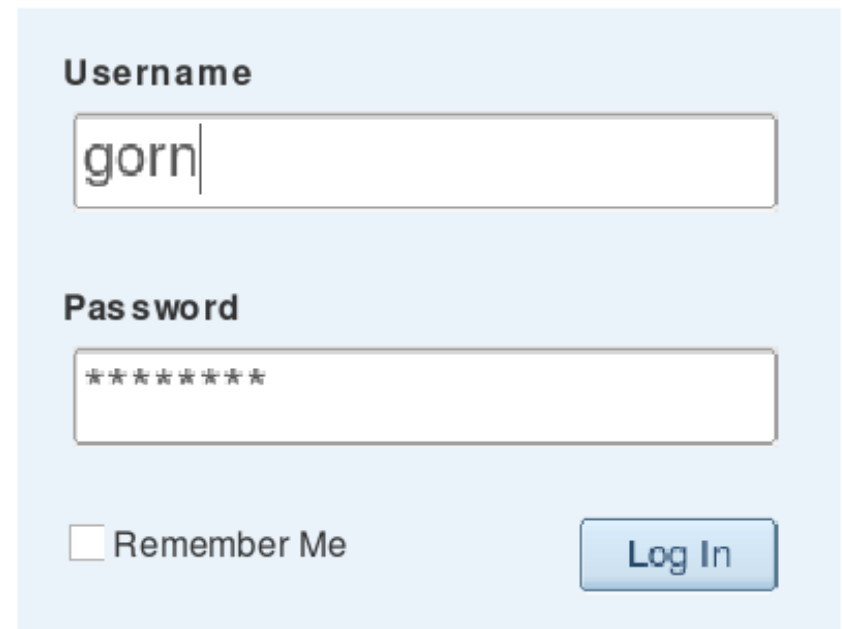
Attach to each object a security label
and type-check the labels against security policy

The Need to Declassify

Software must maintain data confidentiality.

But some programs **need** to leak:

- Password check
- Statistical reports
- Financial transaction

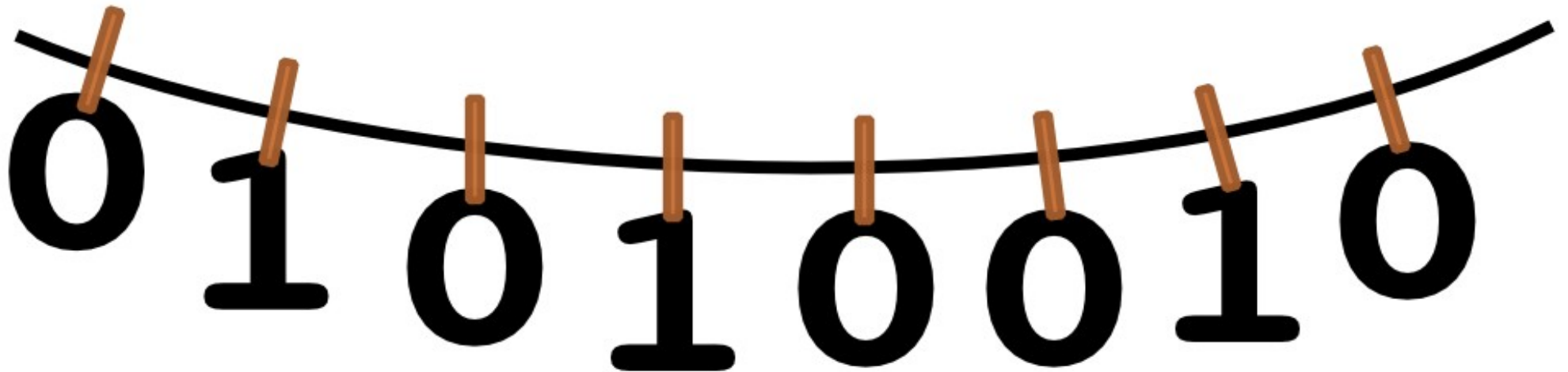


A login form with a light blue background. It contains the following elements:

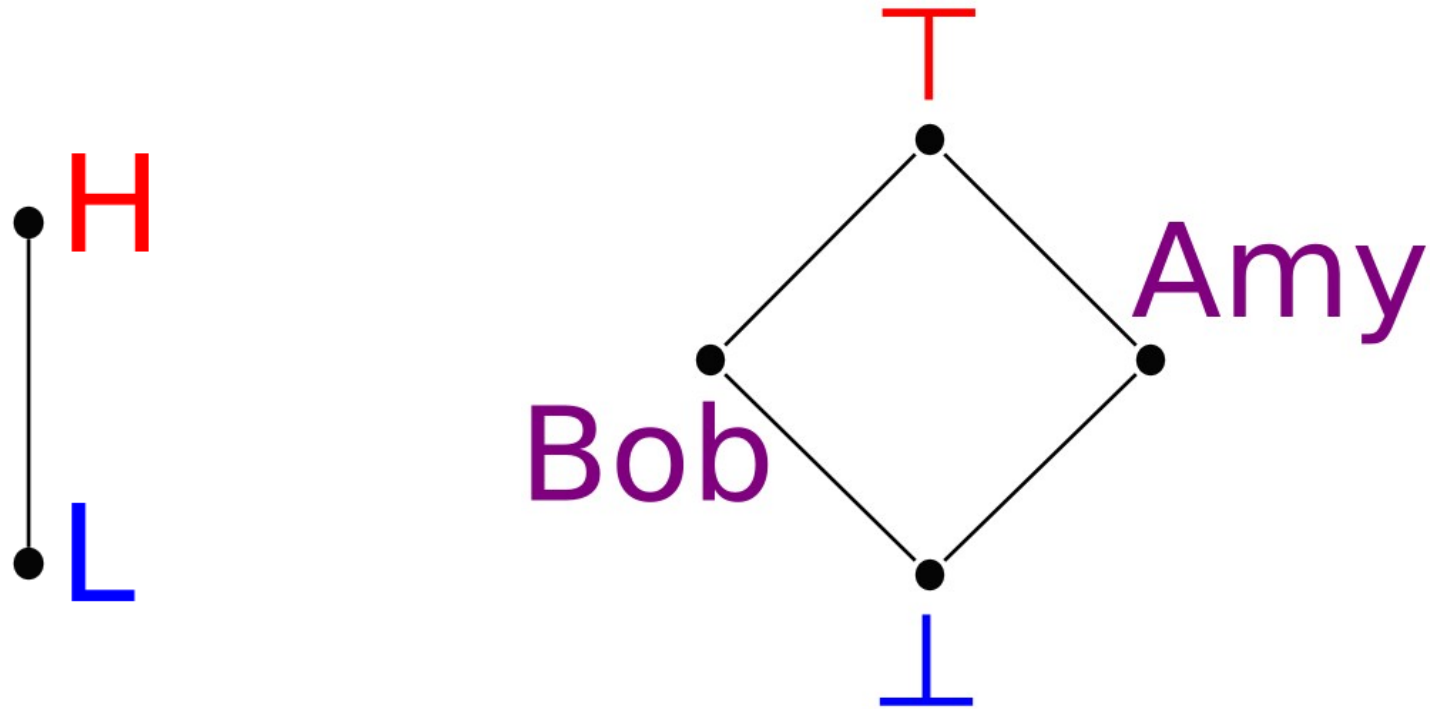
- Username**: A text input field containing the text "gorn".
- Password**: A text input field containing ten asterisks "*****".
- Remember Me: A checkbox with the text "Remember Me" to its right.
- Log In**: A blue button with the text "Log In" in white.

What Might go Wrong

- Might compromise confidentiality
- Violates non-interference
- Secrets can be laundered
- Declassification code has trust blanket



Security Lattice



Principal of Noninterference:

Information can flow from Bob to Amy if
 $\text{Bob} \sqsubseteq \text{Amy}$

Formal Noninterference

Changes in **high** security variables should not be reflected in **low** security variables

$$\forall M_1, M_2 . M_1 \approx_{\ell} M_2 \Rightarrow \langle M_1, c \rangle \approx_{\ell} \langle M_2, c \rangle$$

Memories are indistinguishable

\Rightarrow

Expression configurations are indistinguishable

Intentional Release

```
H = array of Salary  
avg := declassify( sum(H) / len(H), low );
```

- Violates Noninterference
- Can we bound how much info is leaked?
- Need an “escape hatch” to delimit and explicitly state what is to be released

Formal Delimited Release

Command c contains declassify expressions:

$\text{declassify}(e_1, \ell_1)$

...

$\text{declassify}(e_n, \ell_n)$

$$\begin{aligned} & \forall \ell, M_1, M_2. (M_1 \stackrel{\ell}{=} M_2 \ \& \ \forall i \{ i \mid \ell_i \sqsubseteq \ell \}. \langle M_1, e_i \rangle \approx \langle M_2, e_i \rangle) \\ & \Rightarrow \\ & \langle M_1, c \rangle \approx_{\ell} \langle M_2, c \rangle \end{aligned}$$

Memories are indistinguishable

Expressions are indistinguishable

Declassification commands are indistinguishable

Security Through Noninterference

If a program satisfies noninterference
then it is secure.

If declassify does not appear in a secure program
then it satisfies noninterference.

Example: Average Salary

Accepted by delimited release:

```
avg:=declassify( sum(H)/len(H), low );
```

```
    n = len(H);  
    tmp:= H1; H1:=H2; ...; Hn=tmp;  
avg:=declassify( (H1+H2+...+Hn)/n, low );
```

Rejected:

```
    H2=H1; ...; Hn=H1;  
avg:=declassify( (H1+H2+...+Hn)/n, low );
```

Average Salary

Why it's Rejected

```

H2 = H1; ...; Hn = H1;
avg := declassify((H1 + H2 + ... + Hn) / n, low);

```

For $\ell = \text{low}$

$$M_1 = M_2$$

$$\langle M_1, (H_1 + H_2 + H_3) / 3 \rangle \approx \langle M_2, (H_1 + H_2 + H_3) / 3 \rangle$$

But

$$\langle M_1, \text{Avg-Attack} \rangle \neq \langle M_2, \text{Avg-Attack} \rangle$$

$$\text{avg} == 2 \quad \text{vs} \quad \text{avg} == 3$$

	M ₁	M ₂
H ₁	2	3
H ₂	3	2
H ₃	0	0

Example: Electronic Wallet

Purchase an item only if enough money

```
if declassify( wallet > cost, low ) then
    wallet := wallet - cost;
    spent := spent + cost;
```

- Fails noninterference
- Passes delimited release

Example: Attacking the Wallet

```
spent := 0;  
n := 32;  
while (n >= 0) do  
  cost := 2n-1;  
  if declassify(wallet > cost, low) then  
    wallet := wallet - cost;  
    spent := spent + cost;  
  endif  
  n := n - 1  
endwhile;
```

Security Typing

Shall not use other **high** variables
in the definition of H

```
H := ...;  
...  
declassify(H, low)
```

```
while do  
  declassify(H, low)  
  ...  
  H := ...;
```

Create a type system such that
if a program is typable
then it satisfies delimited release and is secure

Example: Password Checking

Results of the hash of password+salt are stored in a publicly viewable database

```
hash(pwd, salt):  $\ell_{\text{pwd}} \times \ell_{\text{salt}} \rightarrow \text{low}$   
= declassify(buildHash(pwd+salt), low)
```

User queries are matched against the stored image

```
match(pwdImg, salt, query):  
     $\ell_{\text{pwdImg}} \times \ell_{\text{salt}} \times \ell_{\text{query}} \rightarrow \ell_{\text{pwdImg}} \sqcup \text{low}$   
= (pwdImg == hash(query, salt));
```

Example: Password Checking

Updating the password

```
update(pwdImg, salt, oldPwd, newPwd)(low  $\sqsubseteq$   $\langle$  pwdImg  $\rangle$ )  
= if match(pwdImg, salt, oldPwd)  
  then pwdImg := hash(newPwd, salt);  
  else skip;
```

The requisite functions
hash, match, update
are all typable,
thus secure

What are the Typings?

```
hash(pwd, salt):  $\ell_{\text{pwd}} \times \ell_{\text{salt}} \rightarrow \text{low}$   
= declassify(buildHash(pwd+salt), low)
```

- Honest user applying hash to password and salt
 $\Gamma \vdash \text{hash}(\text{pwd}, \text{salt}):$
 $\text{high} \times \text{low} \rightarrow \text{low}$
- Attacker hashing a password with honest user's salt
 $\Gamma \vdash \text{hash}(\text{pwd}, \text{salt}):$
 $\text{low} \times \text{low} \rightarrow \text{low}$

What are the Typings?

```
match(pwdImg, salt, query):  
    ℓpwdImg × ℓsalt × ℓquery -> ℓpwdImg ⊔ low  
= (pwdImg == hash(query, salt));
```

- Honest user matching a password

$\Gamma \vdash \text{match}(\text{pwdImg}, \text{salt}, \text{query}):$
 $\text{low} \times \text{low} \times \text{high} \rightarrow \text{low}$

- Attacker tries to guess users password by matching

$\Gamma \vdash \text{match}(\text{pwdImg}, \text{salt}, \text{query}):$
 $\text{low} \times \text{low} \times \text{low} \rightarrow \text{low}$

What are the Typings?

```
update(pwdImg, salt, oldPwd, newPwd)(low  $\sqsubseteq$   $\ell_{\text{pwdImg}}$ )  
  = if match(pwdImg, salt, oldPwd)  
    then pwdImg := hash(newPwd, salt);  
    else skip;
```

- Honest user modifies password
 $\Gamma, \text{low} \vdash \text{update}(\text{pwdImg}, \text{salt}, \text{oldPwd}, \text{newPwd})$:
 $\text{low} \times \text{low} \times \text{high} \times \text{high}$
- Attacker tries to modify users password
 $\Gamma, \text{low} \vdash \text{update}(\text{pwdImg}, \text{salt}, \text{oldPwd}, \text{newPwd})$:
 $\text{low} \times \text{low} \times \text{low} \times \text{low}$

Prevent Password Laundering

```
L := 0;
n := 32;
while (n >= 0) do
  k := 2n-1;
  if hash(sign(h - k + 1), salt) == hash(1, salt)
  then
    h := h - k
    L := L + k
  endif
  n := n - 1
endwhile;
```

for reasonable hash
 $h \geq k \Leftrightarrow h - k + 1 > 0$

Is reject by the type system because
h appears in the declassify
and is updated in the loop body

Related Work

- Selective Independence [Cohen]
- How ?
 - Intransitive Interference [Mantel and Sands]
 - Selective Declassification [Myers and Liskov]
- How much ?
 - Syntax inference [Clark et al.]
 - Approximate noninterference [Di Pierro et al.]
 - Channel capacity [Lowe]
- Relative to what?
 - Leaking complexity [Volpano, Smith, Abadi, Summi, Pierce, Laud]
 - Admissibility [Dam and Giambiagi]
 - Robust Declassification [Myers, Zdancewic]

Conclusion

Capture **what** rather than **how**

Can be integrated with existing security typed languages to build components for delimited release.

Typable => secure

A Small Syntax Problem

```
h:=parity(h);  
if declassify(h==1, low) then (L:=1;h:=1)  
                             else (L:=0;h:=0)
```

Is insecure and rejected by type system

Programmer must write this instead:

```
if declassify(parity(h)==1, low)  
             then (L:=1;h:=1)  
             else (L:=0;h:=0)
```