Enforcing Robust Declassification

Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic


(presented for CMSC838z Spring 2005 by Tim Fraser)
It’s all broken!

- Noninterference + declassification = no assurance
  “Just because it type checks doesn’t mean it keeps secrets.”

- Robust Declassification is a solution
  “When is it OK to declassify?”

- Enforcing R.D. with a type system

- R.D.’s little brother “Qualified Robustness”

- A little history...
How (not) to type check (1:2)

```c
int
main(void) {
    char *p = "l33t!";
    return p();
}
```

```
snow: tfraser$ gcc leet1.c
leet1.c: In function ‘main’:  
leet1.c:5: error: called object is not a function
```
int main(void) {

    char *p = "l33t!";
    return ((int (*)(void))p)();
}

snow: tfraser$ gcc leet2.c
snow: tfraser$ ./a.out
Illegal instruction
Information flow typecheck

secret input;
public output;

\[
\text{temp;}
\]

\[
\text{temp = input; } \quad // \quad s = s
\]

\[
\text{output = temp; } \quad // \quad p = s :^\wedge( !!! \quad \text{public}
\]
Misuse of declassify(), or...

secret input;
public output;

    temp;

temp = input;    // s = s
output = declassify(temp);    // p = p :^D
... a valid use?

secret input;
public output;
    temp;

    temp = input;  // s = s
    temp = encrypt(temp);  // s = s :^o
output = declassify(temp);  // p = p

- types help programmer find problems
- but provide no assurance he resolved them correctly
When can you use declassify?

+ Never!! (strong assurance, but impractical)

+ "Robust Declassification"

+ When programmer thinks its proper

+ Declassify everything!! (weak assurance)
Robust Declassification, informally:

- Declassify only when attackers can’t use low-integrity inputs to trick the program into declassifying.

- “A system is secure if an active attacker (who can observe and modify a part of the system state) may not learn more sensitive information than a passive attacker (who can merely observe visible memories).”
Robust declassification examples (1:2)

```plaintext
public-clean x, y;
secret-clean z;

// program 1:
x = declassify(z);

// program 2:
if x clean
then y = declassify(z);
else skip;
```

Confidentiality:
secret

Integrity:
clean

dirty
Robust declassification examples (2:2)

```plaintext
public-dirty a,b;
secret-clean c;

// program 3:
if a
    then b = declassify(c);
else skip;
```
Outline of formalism

1. Define lattices and $C \times I$ types
2. Define simple imperative language
3. Define semantics with an abstract machine
4. Redefine noninterference in terms of distinguishing machine states
5. Model attacks as added program statements (???)
6. Apply typing rules as usual, plus declassification
Threat model

public-dirty x, y;
secret-clean z;

x = 0;
[*];  // x = 1, x = 2, ...
if x
    then y = declassify(z);
else skip;
Declassification typing rule

Expression $e$ has type $l'$, perhaps secret-clean
Put result in $v$ with type $l$, perhaps public-clean

$$\Gamma, pc \vdash e : l'$$

let’s name my context’s type $l'$

$$l \sqcup pc \subseteq \Gamma(v)$$

declassify decreases confidentiality

$$I(l) = I(l')$$

declassify doesn’t change integrity

$$I(pc), I(l') \in H_I$$

attacker’s can’t get my context’s type

$$\Gamma, pc \vdash v := \text{declassify}(e, l)$$
Qualified robustness (1:2)

public-dirty a;
public-clean b,c;
secret-clean d;

[*]; // a = 0, a = 1, ...
b = endorse(a);
if b
  then c = declassify(d);
else skip;
Qualified robustness (2:2)

public-dirty a, x;
public-clean b, c;
secret-clean d;

[*];  // x = 0, x = 1, ...
if x  // :^(
    then b = endorse(a);
else skip;  // note no b = a here
if b
    then c = declassify(d);
else skip;
Where do we go from here?

+ Never!! (strong assurance, but impractical)
  |
  |
  + What goes here???  (R.D. == correct?)
  |
  |
  + "Robust Declassification"
  |
  |
  + When programmer thinks its proper
  |
  |
  + Declassify everything!! (weak assurance)
Once upon a time...

System Z


1. Start machine
2. Declassify everything to lowest level
3. Run... with no more secrets.