



Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Feedback oriented security analysis

Jeroen Weijers

Departement of Information and Computing Sciences,
Utrecht University

e-mail: jweijers@cs.uu.nl

Thesis Defence

April 1, 2010

Introduction



Writing secure programs

Writing a program that handles secure data is difficult

- ▶ No sensitive data may leak to non secure functions
- ▶ Control flow structures can indirectly leak information



Writing secure programs

Writing a program that handles secure data is difficult

- ▶ No sensitive data may leak to non secure functions
- ▶ Control flow structures can indirectly leak information

The solution: Security Analysis!

- ▶ Static guarantee on confidentiality
- ▶ Secure data labelled as secure
- ▶ Explicitly mark expected security level at outputs



Security Analysis

Security analysis is a validating analysis

- ▶ Type analysis is also a validating analyses
- ▶ Inconsistent programs are rejected
 - ▶ A diagnoses of the problem is given



Example

```
(if (True :: BoolH)  
  then (True :: BoolL)  
  else (False :: BoolL)) :: BoolL
```



Example

H Secure value!

```
(if (True :: BoolH)
  then (True :: BoolL)
  else (False :: BoolL)) :: BoolL
```



Example

H Secure value!

L Non secure value!

L Non secure value!

```
(if (True :: BoolH)  
  then (True :: BoolL)  
  else (False :: BoolL)) :: BoolL
```



Example

H Secure value!

L Non secure value!

L Non secure value!

```
(if (True :: BoolH)  
  then (True :: BoolL)  
  else (False :: BoolL)) :: BoolL
```

L Non secure value!



Example

H Secure value!

L Non secure value!

L Non secure value!

```
(if (True :: BoolH)  
  then (True :: BoolL)  
  else (False :: BoolL)) :: BoolL
```

Outcome of conditional is revealed!



Example

H Secure value!

L Non secure value!

L Non secure value!

```
(if (True :: BoolH)  
  then (True :: BoolL)  
  else (False :: BoolL)) :: BoolL
```

Outcome of conditional is revealed!

- ☞ Programs that leak secure information should be rejected by the compiler!



Polyvariant security analysis

$$lowId :: \forall \alpha. \alpha^L \rightarrow \alpha^L$$
$$lowId\ x = x$$
$$highId :: \forall \alpha. \alpha^H \rightarrow \alpha^H$$
$$highId\ x = x$$


Polyvariant security analysis

$$\text{lowId} :: \forall \alpha. \alpha^L \rightarrow \alpha^L$$
$$\text{lowId } x = x$$
$$\text{highId} :: \forall \alpha. \alpha^H \rightarrow \alpha^H$$
$$\text{highId } x = x$$

With polyvariance:

$$\text{id} :: \forall \alpha. \forall \beta. \alpha^\beta \rightarrow \alpha^\beta$$
$$\text{id } x = x$$


Providing feedback

Providing feedback for security analysis

- ▶ Polyvariance leads to propagation
 - ▶ Security annotations are propagated by polyvariant functions
 - ▶ Mistakes are also propagated
- ▶ Security annotations are only inferred
- ▶ Different from type errors
 - ▶ Due to explicit type signatures type errors are more local
 - ▶ Functions manipulate data and their types
 - ▶ Most functions don't change security levels



FlowCaml

- ▶ FlowCaml is an OCaml dialect from Pottier and Simonet
- ▶ Extends OCaml with a security system
- ▶ Declassification through regular OCaml
 - ▶ Declassification decreases a security level
- ▶ Security levels defined in explicit types



FlowCaml Example

```
flow ! everyone < !root  
val checkPwd : 'β string →  
    (!everyone, !everyone) pwdEntry → 'β bool  
val pwdFile : (!everyone, !root) pwdEntry  
let login pwd = ... checkPwd pwd pwdFile
```



FlowCaml Example

```
flow ! everyone < !root  
val checkPwd : 'β string →  
    (!everyone, !everyone) pwdEntry → 'β bool  
val pwdFile : (!everyone, !root) pwdEntry  
let login pwd = ... checkPwd pwd pwdFile
```

```
File "Login.fml", line 11, characters 0-144:  
This expression generates the following information  
flow:  
!root < !everyone  
which is not legal.
```



Haskell Library by Russo, Claessen and Hughes

- ▶ Uses Haskell type system to enforce security
- ▶ Security implemented with type classes and monads
- ▶ Illegal flows are reported as regular type errors
- ▶ Controlled explicit declassification



Haskell Library Example

```
verifyPassword :: Less  $\alpha$  R  $\Rightarrow$   
  Sec R Spwd  $\rightarrow$  Sec  $\alpha$  Pwd  $\rightarrow$  Sec  $\alpha$  Bool  
verifyPassword record provided =  
  let known = fromJust $ ((hatch ( $\lambda(-, p) \rightarrow p$ ) record)  
    :: Maybe (Sec R Pwd))  
  in combine ( $\equiv$ ) known provided
```



Haskell Library Example

Spwd.hs:17:13:

Could not deduce (Less R a) from the context (Less a R)
arising from a use of 'combine' at Spwd.hs
:17:13-39

Possible fix:

add (Less R a) to the context of
the type signature for 'verifyPassword'
or add an instance declaration for (Less R a)

In the expression: combine (==) known provided

In the expression:

let

known = fromJust

\$ ((hatch \ (_, p) -> p) record)

:: Maybe (Sec R Pwd)

in combine (==) known provided

In the definition of 'verifyPassword':

verifyPassword record provided

= let

known = fromJust

\$ ((hatch \ (_, p) -> ...)

Information and Computing Sciences



Our error message

```
Error in application:  
"(checkPwd 1) password" at (line 13, column 9)  
The function: "(checkPwd 1)"  
Expected an argument protected at at most level: Low  
But the given argument: "password"  
Is protected at a higher level.
```



Outline

The aim of this work is:

Gain more control over security error messages.

- ▶ The source language
- ▶ Security analysis
- ▶ Generating error messages



Source language



The source language

- ▶ Simply typed, lambda calculus based language
- ▶ Call by value semantics
- ▶ Recursive functions **fun** $f x \Rightarrow e_1$
- ▶ Special constructs for security analysis
 - ▶ **protect** e_0 s increases security level
 - ▶ **declassify** e_0 s decreases security level



protect

The protect statements increases the security level of an expression

$$\textit{normalValue} = 1$$


protect

The protect statements increases the security level of an expression

$$\mathit{normalValue} = 1$$
$$\mathit{secureValue} = \mathbf{protect} \mathit{normalValue} \mathbf{H}$$


protect

The protect statements increases the security level of an expression

$$\mathit{normalValue} = 1$$
$$\mathit{secureValue} = \mathbf{protect} \mathit{normalValue} \mathbf{H}$$
$$\mathit{nonValid} = \mathbf{protect} \mathit{secureValue} \mathbf{L}$$


declassify

The declassify statements decreases the security level of an expression

secure Value = **protect 1 H**



declassify

The declassify statements decreases the security level of an expression

$$\text{secure Value} = \text{protect } 1 \text{ H}$$
$$\text{nonSecure} = \text{declassify } \text{secure Value } \text{L}$$


declassify

The declassify statements decreases the security level of an expression

$$\textit{secure Value} = \text{protect } 1 \text{ H}$$
$$\textit{nonSecure} = \text{declassify } \textit{secure Value} \text{ L}$$
$$\textit{nonValid} = \text{declassify } \textit{nonSecure} \text{ H}$$


Subeffecting

$$sIncr :: \text{Int}^H \rightarrow \text{Int}^H$$
$$val :: \text{Int}^L$$

Is it safe to apply *sIncr* to *val*?



Subeffecting

$$sIncr :: \text{Int}^H \rightarrow \text{Int}^H$$
$$val :: \text{Int}^L$$

Is it safe to apply *sIncr* to *val*?

Yes, there is no danger in treating a value as more secure than it is!

The type of *sIncr val* is Int^H .

☞ Any security level can silently be cast to a higher level



Security Analysis



Type and effect system

- ▶ Let polymorphism
- ▶ Polyvariant constraint based security analysis
- ▶ Subeffecting through constraints
- ▶ Based on the FlowCaml system extended with explicit declassification



Security levels

Security levels l are defined in a lattice \mathcal{L} .

This lattice has the properties:

- ▶ There exists a bottom and top element element
- ▶ $x \sqcup y$ and $x \sqcap y$ exists for all $x, y \in \mathcal{L}$



Security levels

Security levels l are defined in a lattice \mathcal{L} .

This lattice has the properties:

- ▶ There exists a bottom and top element element
- ▶ $x \sqcup y$ and $x \sqcap y$ exists for all $x, y \in \mathcal{L}$

The simplest security lattice is the two point lattice with:

- ▶ L for \perp
- ▶ H for \top



Type language

Our type language is defined as:

$$\begin{aligned}\tau &::= \text{Int} \mid \text{Bool} \mid \text{List } \tau^\varphi \mid (\tau_1^{\varphi_1}, \tau_2^{\varphi_2}) \\ &\quad \mid \tau_1^{\varphi_1} \rightarrow \tau_2^{\varphi_2} \mid \alpha \\ \varphi &::= \beta \mid l \\ \pi &::= \varphi_1 \sqsubseteq \varphi_2 \\ \sigma &::= \forall \alpha. \sigma \mid \forall \beta. \sigma \mid \tau \\ \Gamma &::= \emptyset \mid \Gamma [x \mapsto (\sigma, \varphi)] \\ C &::= \emptyset \mid \{\pi_1, \dots, \pi_n\} \cup C\end{aligned}$$



Inference rules

The type system is defined as a set of rules with judgements of the form:

$$\Gamma, C \vdash e : \sigma^\varphi$$

Which is read as: under the type and constraint environments Γ and C expression e is assigned a type scheme σ with security level φ .



Inference rules

$$\frac{\begin{array}{l} \Gamma, C \vdash e_0 : \mathbf{Bool}^{\varphi_0} \\ \Gamma, C \vdash e_1 : \tau^{\varphi_1} \\ \Gamma, C \vdash e_2 : \tau^{\varphi_2} \\ C \vdash \varphi_0 \sqsubseteq \varphi \quad C \vdash \varphi_1 \sqsubseteq \varphi \quad C \vdash \varphi_2 \sqsubseteq \varphi \end{array}}{\Gamma, C \vdash \mathbf{if } e_0 \mathbf{ then } e_1 \mathbf{ else } e_2 : \tau^{\varphi}} \quad [t\text{-if}]$$



Inference rules

$$\frac{\begin{array}{l} \Gamma, C' \vdash e_1 : \tau_1^{\varphi_1} \\ (C'', \sigma) = \text{gen}(C', \tau_1) \\ \Gamma [x \mapsto (\sigma, \varphi_1)], C \cup C'' \vdash e_2 : \tau_2^{\varphi_2} \\ C \cup C'' \vdash (\varphi_2 \sqsubseteq \varphi_3) \end{array}}{\Gamma, C \cup C'' \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau_2^{\varphi_3}} \quad [t\text{-let}]$$



Inference rules

$$\frac{\begin{array}{l} \Gamma, C' \vdash e_1 : \tau_1^{\varphi_1} \\ (C'', \sigma) = \text{gen}(C', \tau_1) \\ \Gamma [x \mapsto (\sigma, \varphi_1)], C \cup C'' \vdash e_2 : \tau_2^{\varphi_2} \\ C \cup C'' \vdash (\varphi_2 \sqsubseteq \varphi_3) \end{array}}{\Gamma, C \cup C'' \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau_2^{\varphi_3}} \quad [t\text{-let}]$$

There are 19 other rules we will not discuss.



Inference algorithm

We implemented an inference algorithm

- ▶ Based on algorithm \mathcal{W}
- ▶ Types are unified along the way
- ▶ Constraints are collected and checked at generalisation
- ▶ Common way of implementing type and effect systems



Constraint satisfiability

Verify constraint satisfiability when generalising

Verification of satisfiability:

- ▶ Initialise for each variable β in the constraints in C a range of values (\perp, \top)
- ▶ Use a worklist algorithm to compute the actual ranges



Constraint satisfiability

Verify constraint satisfiability when generalising

Verification of satisfiability:

- ▶ Initialise for each variable β in the constraints in C a range of values (\perp, \top)
- ▶ Use a worklist algorithm to compute the actual ranges

A constraint set is satisfiable if: For all variables $\beta \in C$ the range $(s1, s2)$ satisfies $s1 \sqsubseteq s2$.

- ☞ If for any variable the range is empty the program contains a security leak!



Generating error messages



When an inconsistency is found

When an inconsistency is found a minimal unsatisfiable set of constraints is computed.

- ▶ These constraints together form a path
- ▶ The end points are conflicting security levels
- ▶ Each constraint is a potential cause of the inconsistency
- ▶ For each constraint its source in the AST is known



When an inconsistency is found

When an inconsistency is found a minimal unsatisfiable set of constraints is computed.

- ▶ These constraints together form a path
- ▶ The end points are conflicting security levels
- ▶ Each constraint is a potential cause of the inconsistency
- ▶ For each constraint its source in the AST is known

The source nodes of the constraints in the minimal unsatisfiable constraint set cover all program points related to the inconsistency.

- ▶ It is already a good error, but it can be a large fragment



Heuristics

Heuristics are used to find the cause of the inconsistency in the minimal unsatisfiable set.

Two sorts of heuristics:

- ▶ Filter heuristics
- ▶ Selector heuristics



Propagation heuristic (filter heuristic)

Most constraints have the form: $\beta_1 \sqsubseteq \beta_2$.

They are generated for subeffecting in polyvariant functions (*id*, *cons*, etc).

And by some control flow structures as **if ... then ... else ...**



Propagation heuristic (filter heuristic)

secureVal :: Int^H

incr :: $\forall \beta_1. \forall \beta_2. \beta_1 \sqsubseteq \beta_2 \Rightarrow \text{Int}^{\beta_1} \rightarrow \text{Int}^{\beta_2}$

id :: $\forall \alpha. \forall \beta_1. \forall \beta_2. \beta_1 \sqsubseteq \beta_2 \Rightarrow \alpha^{\beta_1} \rightarrow \alpha^{\beta_2}$

print :: $\forall \alpha. \alpha^L \rightarrow \alpha^L$

print (*incr* (*incr* (*id* *secureVal*)))



Propagation heuristic (filter heuristic)

```
secureVal :: IntH  
incr :: ∀β1. ∀β2. β1 ⊆ β2 ⇒ Intβ1 → Intβ2  
id :: ∀α. ∀β1. ∀β2. β1 ⊆ β2 ⇒ αβ1 → αβ2  
print :: ∀α. αL → αL  
print (incr (incr (id secureVal)))
```

No evidence that any propagating function is more wrong than others

Blaming one is an arbitrary choice



Propagation heuristic (filter heuristic)

```
secureVal :: IntH  
incr :: ∀β1.∀β2. β1 ⊆ β2 ⇒ Intβ1 → Intβ2  
id :: ∀α.∀β1.∀β2. β1 ⊆ β2 ⇒ αβ1 → αβ2  
print :: ∀α. αL → αL  
print (incr (incr (id secureVal)))
```

No evidence that any propagating function is more wrong than others

Blaming one is an arbitrary choice

The propagation heuristic removes propagation constraints from the set of candidates.



Other heuristics

We defined several other heuristics:

- ▶ Irrefutable constraints heuristic (filter heuristic)
- ▶ Sibling heuristic (selector heuristic)
- ▶ Majority heuristic (selector heuristic)
- ▶ Least trusted constraint heuristic (selector heuristic)



Example

```
print ::  $\forall \alpha. \alpha^L \rightarrow \alpha^L$   
passwordFile ::  $\forall \beta_1. \forall \beta_2. (\text{List } (\text{Int}^{\beta_1}, \text{Int}^H)^{\beta_2})^L$   
login = fn u  $\Rightarrow$  fn p  $\Rightarrow$   
  print (let userRecord = (findUser u passwordFile)  
    in (if (null userRecord)  
      then False  
      else ((snd (hd userRecord))  $\equiv$  p)))
```



Example

The least trusted constraint heuristic generates:

```
Error in application:
```

```
"(print let userRecord = ((findUser u) passwordFile)
  in if null userRecord then False else (snd head
    userRecord == p))" at: (line 12, column 25)
```

```
The function: "print"
```

```
Expected an argument protected at at most level: Low
```

```
But the given argument: "let userRecord = ((findUser
  u) passwordFile) in if null userRecord then
  False else (snd head userRecord == p)"
```

```
Is protected at a higher level.
```



Default error message

If the heuristics can not find the cause a program slice is presented.

```
Sort of error : use of secure value as less secure
argument , endpoints Low vs . High
print (let (...)
      in (if (...)
          then (...)
          else ((snd (hd userRecord)) == (...))))
```



Conclusions



Conclusions

Defined and implemented a constraint based security analysis

- ▶ Security errors are discovered separate from type errors

Program slices give rise to accurate error messages

- ▶ But can be large

We gained control over error messages by using constraints

- ▶ We apply heuristics to these constraints:
 - ▶ to filter unlikely causes
 - ▶ to find common mistakes
 - ▶ to blame the most likely cause



Future work

- ▶ Specialised heuristics for program constructs
 - ▶ For example: The programmer forgot that the security level of e_0 in **if** e_0 **then** e_1 **else** e_2 also contributes to the overall security level.
- ▶ Explicit security type signatures
- ▶ Constraint solving over a larger scope
- ▶ Hints for optimising analysis in the source program

