Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Security Type Error Diagnosis

Jurriaan Hage

Joint work with Jeroen Weijers and Stefan Holdermans

Department of Information and Computing Sciences, Universiteit Utrecht
J.Hage@uu.nl

February 26, 2013

Some things about me

- PhD at Uni. Leiden under Grzegorz Rozenberg on algorithms and combinatorics of graphs and groups (switching classes)
- Commercial educator at Leiden during 1999-2000
- With Doaitse Swierstra (Johan Jeuring, etc.) since Nov 2000
- Topics of interest:
  - static analysis and software analysis
  - mostly functional languages
  - plagiarism detection
  - "testing"
  - "SOA"
  - type error feedback
- I try to frequent conferences like POPL, ICFP, PEPM

Blatant advertisement

- Next year, I chair
  - PEPM 2014, San Diego, co-located with POPL
  - TFP 2014, somewhere in the Netherlands
- Start saving up papers to submit!

Type error diagnosis

- Slogan of the static analyst: bigger, faster, more (precise)
- Complicated languages and complicated analyses are difficult to communicate about
- Plenty of work on type error diagnosis for the polymorphic lambda-calculus
- But little or nothing on other validating analyses
- In this talk: Security Analysis (a la Volpano and Smith)

Type error diagnosis

- Slogan of the static analyst: bigger, faster, more (precise)
- Complicated languages and complicated analyses are difficult to communicate about
- Plenty of work on type error diagnosis for the polymorphic lambda-calculus
- But little or nothing on other validating analyses
- In this talk: Security Analysis (a la Volpano and Smith)
- Caveat: this talk is not really about security analysis

Security Analysis

- A security type system can be imposed on a (functional) language to ensure that programs preserve confidentality
- $\bot$ line: highly confidential values may not flow to not so confidential locations (in the talk: $\mathsf{L} \sqsubset \mathsf{H}$)
- Example:

$$
\begin{aligned}
&\mathit{showLow} :: \alpha^{\mathsf{L}} \to \mathit{IO}\ ()^{\mathsf{L}} \\
&\mathit{showLow} \\
&\quad (\mathbf{if}\ (\mathit{True} :: \mathtt{Bool}^{\mathsf{H}})\ \mathbf{then}\ (\mathit{True} :: \mathtt{Bool}^{\mathsf{L}}) \\
&\qquad \mathbf{else}\ (\mathit{False} :: \mathtt{Bool}^{\mathsf{L}}) :: \mathtt{Bool}^{\mathsf{H}})
\end{aligned}
$$

- To write practical programs we need **declassify**, just as you need $\mathit{unsafePerformIO}$ in Haskell

Security type system

- Implemented as a non-standard type system (annotations on top of the intrinsic type system)
- Security Analysis is an instance of a dependency analysis (Abadi99)
  - Therefore, it deviates in some respects from the intrinsic type system
- Type system is "state of the art":
  - subeffecting (but no full subtyping),
  - polyvariant (but no higher-ranked polyvariance)
  - underlying language polymorphic

The subject language (a la FlowCaml)

- A program consists of declarations $f = e$
- Expressions consist of
  - numbers, booleans, variables
  - functions: **fn** $x \Rightarrow e_0$ and **fun** $f\ x \Rightarrow e_0$
  - application: $e_0\ e_1$
  - conditionals: $\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$
  - recursive let $\mathbf{let}\ x = e_0\ \mathbf{in}\ e_1$
  - unary and binary operators: $e_1 \oplus e_2$ and $u\ e_1$
  - built-in lists and pairs: **Cons** $e_1\ e_2$, **Nil** and (e_1, e_2)
  - deconstructors: **fst** e_1, **snd** e_1, **null** e_1, **hd** e_1, and **tl** e_1.
  - unsafe-perform-declassify: **declassify** $e_0\ s$
    (s must be at most as confidential as e_0)
  - To increase confidentiality: **protect** $e_0\ s$
    (s must be at least as confidential as e_0)
- s ranges over any complete confidentiality lattice

Annotate the types

- Annotated types τ^{φ} where φ a security level or variable, and τ taken from
  - `Int`
  - `Bool`
  - `List` τ^{φ}
  - $({\tau_1}^{\varphi}, {\tau_2}^{\varphi})$, and
  - ${\tau_1}^{\varphi} \to {\tau_2}^{\varphi}$
- The spine of a list may be protected at H while the elements are of low confidentiality: $(\texttt{List}\ \alpha^{\mathsf{L}})^{\mathsf{H}}$
- Generalisation over type variables and annotation variables
  - Polymorphic and polyvariant
- Constraint solving takes place per binding group.

A few type rules

$$\frac{\Gamma, C \vdash e_0 : \texttt{Bool}^{\varphi_1} \quad \Gamma, C \vdash e_1 : \tau^{\varphi_2} \quad \Gamma, C \vdash e_2 : \tau^{\varphi_3}}{\Gamma, C \vdash \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 : \tau^{\varphi_1 \sqcup \varphi_2 \sqcup \varphi_3}} \; [\textit{t-if}]$$

$$\frac{\Gamma, C \vdash e_1 : (\texttt{List } \tau^{\varphi_1})^{\varphi}}{\Gamma, C \vdash \textbf{hd } e_1 : \tau^{\varphi_1 \sqcup \varphi}} \; [\textit{t-hd}]$$

$$\frac{\Gamma, C \vdash e : \tau^{\varphi} \quad C \vdash \varphi_0 \sqsubseteq \varphi}{\Gamma, C \vdash \textbf{declassify } e \; \varphi_0 : \tau^{\varphi_0}} \; [\textit{t-declass}]$$

$$\frac{\Gamma, C \vdash e : \tau^{\varphi} \quad C \vdash \varphi \sqsubseteq \varphi_0}{\Gamma, C \vdash \textbf{protect } e \; \varphi_0 : \tau^{\varphi_0}} \; [\textit{t-protect}]$$

Approach

0 Implement type system by generating (security) constraints, solve by fixpoint iteration

Approach

0 Implement type system by generating (security) constraints, solve by fixpoint iteration

- If constraints consistent, then done, otherwise

1 compute a security type error slice.

- Everything outside the slice can be safely ignored
  - And must not play any role in diagnosis

Approach

0 Implement type system by generating (security) constraints, solve by fixpoint iteration

- If constraints consistent, then done, otherwise

1 compute a security type error slice.

- Everything outside the slice can be safely ignored
  - And must not play any role in diagnosis

2 Apply various heuristics to constraints in the slice to
   a further narrow down locations to the more likely causes
   b suggest fixes for the mistake

- Construct error report from the remaining constraints, and which heuristics "fired"

Step 0: the move to constraints

- Type system implemented as the solving of a set of constraints, generated from the program
  $\beta \sqsubseteq \mathsf{H}$, $\beta_1 \sqsubseteq \beta_2$, $\mathsf{H} \sqsubseteq \mathsf{L}$
- Essentially: $\beta_1 \sqsubseteq \beta_2$ means that β_2 is at least as confidential as β_1
- $\beta \sqsubseteq \mathsf{H}$ is a tautology, $\mathsf{H} \sqsubseteq \mathsf{L}$ is in fact unsatisfiable, and for $\beta_1 \sqsubseteq \beta_2$ it all depends on what we know about β_1 and β_2
- Refactor type rules (a bit)

Toward constraints

$$\frac{\Gamma, C \vdash e_0 : \texttt{Bool}^{\varphi_1} \quad \Gamma, C \vdash e_1 : \tau^{\varphi_2} \quad \Gamma, C \vdash e_2 : \tau^{\varphi_3}}{\Gamma, C \vdash \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 : \tau^{\varphi_1 \sqcup \varphi_2 \sqcup \varphi_3}} \; [\textit{t-if}]$$

becomes

$$\frac{\Gamma, C \vdash e_0 : \texttt{Bool}^{\varphi_0} \quad \Gamma, C \vdash e_1 : \tau^{\varphi_1} \quad \Gamma, C \vdash e_2 : \tau^{\varphi_2}}{\Gamma, C \vdash \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 : \tau^{\varphi}} \; [\textit{t-if}]$$

where additionally $\varphi_0 \sqsubseteq \varphi$, $\varphi_1 \sqsubseteq \varphi$, and $\varphi_2 \sqsubseteq \varphi$
Typically: each expression gets a fresh annotation variable β.

Some words on solving constraints

- To achieve polyvariance, solving takes place just before generalisation, i.o.w., per binding group. To find out...
  - which annotation variables should be generalized
  - which should be existentially quantified
    - They only play a role locally
  - which should be left alone
- To every annotation variable, we associate a lowest and highest confidentiality $[\varphi_0, \varphi_1]$
- Some constraints push up φ_0, others push down φ_1
- If φ_1 ends up under φ_0, we have caught an inconsistency.

Step 1: security type error slicing

- Based on Haack and Wells (ICFP 2003/SciCom 2004) and Stuckey et al (Haskell 2003)
- Computes all locations in the program that generate constraints that contribute to the inconsistency of the constraint set
- Short algorithm but not trivial

Step 2: apply heuristics

- Step 1 delivers a set of constraints
- It may be large, and a slice does not provide suggestions and reasons for inconsistencies
- We apply a series of heuristics to arrive at a more precise diagnosis in many cases

Step 2: apply heuristics

- Step 1 delivers a set of constraints
- It may be large, and a slice does not provide suggestions and reasons for inconsistencies
- We apply a series of heuristics to arrive at a more precise diagnosis in many cases
- but at the risk of being wrong!

Classes of heuristics

1. generic heuristics, e.g., a majority heuristic and avoid irrefutable constraints

Classes of heuristics

1. generic heuristics, e.g., a majority heuristic and avoid irrefutable constraints
2. the propagation heuristics: do not blame security-agnostic functions that only propagate security levels

Classes of heuristics

1. generic heuristics, e.g., a majority heuristic and avoid irrefutable constraints
2. the propagation heuristics: do not blame security-agnostic functions that only propagate security levels
3. security specific heuristics: confusing **protect** and **declassify**

Classes of heuristics

1. generic heuristics, e.g., a majority heuristic and avoid irrefutable constraints
2. the propagation heuristics: do not blame security-agnostic functions that only propagate security levels
3. security specific heuristics: confusing **protect** and **declassify**
4. dependency analysis specific heuristics: up next...

The everyday functional programmer...

- .... may be used to the intrinsic type system, but less used to security types
- Intuitions of intrinsic type do not always carry over to security type system
- Inventory of differences between Hindley-Milner and the security type system
- For each such difference we implemented a heuristic to discover whether an inconsistency can be explained from the difference

Back to the example (almost)

$$showLow :: \alpha^{\mathsf{L}} \to IO\ ()^{\mathsf{L}}$$

$$\begin{array}{l} showLow \\ \quad (\mathbf{if}\ (True :: \mathtt{Bool}^{\mathsf{H}})\ \mathbf{then}\ (True :: \mathtt{Bool}^{\mathsf{L}}) \\ \qquad \mathbf{else}\ (False :: \mathtt{Bool}^{\mathsf{L}}) :: \mathtt{Bool}^{\mathsf{L}}) \end{array}$$

The mistake may be that the H on the condition is ignored, because the value of the condition does not flow to the result of the conditional. But the value of the conditional does depend on the value of the boolean!

Back to the rules

$$\frac{\Gamma, C \vdash e_0 : \texttt{Bool}^{\varphi_1} \quad \Gamma, C \vdash e_1 : \tau^{\varphi_2} \quad \Gamma, C \vdash e_2 : \tau^{\varphi_3}}{\Gamma, C \vdash \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 : \tau^{\varphi_1 \sqcup \varphi_2 \sqcup \varphi_3}} \; [\textit{t-if}]$$

The give away: `Bool` does not influence τ, but φ_1 does influence the annotation on τ

Back to the rules

$$\frac{\Gamma, C \vdash e_0 : \texttt{Bool}^{\varphi_1} \quad \Gamma, C \vdash e_1 : \tau^{\varphi_2} \quad \Gamma, C \vdash e_2 : \tau^{\varphi_3}}{\Gamma, C \vdash \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 : \tau^{\varphi_1 \sqcup \varphi_2 \sqcup \varphi_3}} \; [\textit{t-if}]$$

The give away: `Bool` does not influence τ, but φ_1 does influence the annotation on τ

$$\frac{\Gamma, C \vdash e_1 : (\texttt{List } \tau^{\varphi_1})^{\varphi}}{\Gamma, C \vdash \textbf{hd } e_1 : \tau^{\varphi_1 \sqcup \varphi}} \; [\textit{t-hd}]$$

Extracting a value from a list also tells us something about the structure of the list: that it is non-empty

The propagation heuristic

$$
\begin{aligned}
&id :: \alpha^{\beta} \to \alpha^{\beta} \\
&showLow :: \alpha^{\mathsf{L}} \to IO\ ()^{\mathsf{L}} \\
&showLow \\
&\quad (\mathbf{if}\ (id\ (True :: \mathtt{Bool}^{\mathsf{H}}))\ \mathbf{then}\ (True :: \mathtt{Bool}^{\mathsf{L}}) \\
&\qquad \mathbf{else}\ (False :: \mathtt{Bool}^{\mathsf{L}}) :: \mathtt{Bool}^{\mathsf{L}})
\end{aligned}
$$

Makes no sense to blame the call to id, so we ignore the constraints that propagate security annotations through security agnostic functions.

A sizable example

$$log = \textbf{fn}\ x \Rightarrow \textbf{protect}\ x\ Low$$
$$boolVal = \textbf{protect}\ True\ Low$$
$$lVal = \textbf{protect}\ 2\ Low$$
$$zl = \textbf{Cons}\ (\textbf{protect}\ 1\ Low)\ (\textbf{protect}\ \textbf{Nil}\ High)$$
$$id = \textbf{fn}\ x \Rightarrow \textbf{let}\ y = x\ \textbf{in}\ y$$
$$main = log\ (\textbf{if}\ id\ (id\ boolVal)\ \textbf{then}\ id\ lVal\ \textbf{else}\ \textbf{hd}\ zl)$$

The structure of list: zl at (l 11, c 40) passed to **hd** in the expression: **hd** zl at (l 11, c 37) is protected at level: $High$. This causes the result of **hd** to be protected at level: $High$. Instead a value protected at level: Low was expected by: log ....

Majority heuristic example

$$
\begin{aligned}
one &= \textbf{protect}\ 1\ Low \\
two &= \textbf{protect}\ 2\ Low \\
three &= \textbf{protect}\ 3\ Low \\
four &= \textbf{protect}\ 4\ Low \\
five &= \textbf{protect}\ 5\ High \\
fifteen &= print\ (one + two + three + four + five)
\end{aligned}
$$

```
Error in application:
```
$(print\ ((((one + two) + three) + four) + five))$
```
                                    at: (l 6, c 12)
Expected an argument protected at at most level:
```
Low
```
The argument is protected at level:
```
$High$
```
Because of the following subexpression(s):
```
$five$ `at: (l 6, c 46)`

Related Work

- FlowCaml (Pottier, Simonet, TOPLAS '03): formally our basis, no attention paid to type error diagnosis

```
<loc>: This expression generates the following
  information flow: root < everyone
  which is not legal.
```

- Jif system (King et al): Java based, provides sliced execution trace, no higher-order functions, no polymorphism, limited polyvariance by duplication
- Russo, Claessen and Hughes (Haskell '08): embedded into Haskell, no separation of type and annotation

Related Work

- FlowCaml (Pottier, Simonet, TOPLAS '03): formally our basis, no attention paid to type error diagnosis

```
<loc>: This expression generates the following
  information flow: root < everyone
  which is not legal.
```

- Jif system (King et al): Java based, provides sliced execution trace, no higher-order functions, no polymorphism, limited polyvariance by duplication
- Russo, Claessen and Hughes (Haskell '08): embedded into Haskell, no separation of type and annotation
  - See my talk on Friday!

Summary

- A superficial impression of
  - how to design a security analysis as a type and effect system
  - four classes of heuristics to specialize the type error diagnosis, after a first approximation computed by type error slicing
  - a few examples to show the results of our work
- First to combine slicing with heuristics
- The paper contains many many more details

Future Work

- How well does our work perform on realistic programs?
  - how often are the heuristics wrong?
  - how well do the heuristics perform without the slicing?
  - how large are the slices anyway?
  - how often does which heuristic contribute to reduce the number of constraints?
  - how often does it reduce the size of remaining constraints to a singleton?

Future Work

- How well does our work perform on realistic programs?
  - how often are the heuristics wrong?
  - how well do the heuristics perform without the slicing?
  - how large are the slices anyway?
  - how often does which heuristic contribute to reduce the number of constraints?
  - how often does it reduce the size of remaining constraints to a singleton?
- Does anybody have a collection of security incorrect programs?

Future Work

- How well does our work perform on realistic programs?
  - how often are the heuristics wrong?
  - how well do the heuristics perform without the slicing?
  - how large are the slices anyway?
  - how often does which heuristic contribute to reduce the number of constraints?
  - how often does it reduce the size of remaining constraints to a singleton?
- Does anybody have a collection of security incorrect programs?
- Security analysis is a validating analysis. Can we give it the higher-ranked polyvariance treatment (Holdermans and Hage, ICFP 2010)?

Thank you for your attention