Type-Driven Gradual Security Typing

Matías Toro, Ronald Garcia, Éric Tanter
Scenario

let age : Int = 31
let salary : Int = 58000
let intToString : Int -> String = ...
let print : String -> Unit = ...
print(intToString(salary))

Disney and Flanagan. “Gradual Information Flow Typing”
Scenario

let age : Int = 31
let salary : Int = 58000
let intToString : Int → String = ...
let print : String → Unit = ...
print(intToString(salary))

Low Security Data
Scenario

```
let age : Int = 31
let salary : Int = 58000
let intToString : Int → String = ...
let print : String → Unit = ...
print(intToString(salary))
```
let age : Int = 31
let salary : Int = 58000
let intToString : Int → String = ...
let print : String → Unit = ...
print(intToString(salary))
let age : Int = 31
let salary : Int = 58000
let intToString : Int → String = ...
let print : String → Unit = ...
print(intToString(salary))
More Types!

Γ ⊩ □
Information-Flow Security Typing
A Lattice Model of Secure Information Flow

Dorothy E. Denning
Purdue University

[CACM 1976]
Security as a Lattice

Classification

Permitted Readers

Zdancewic. “Programming Languages for Information Security”
Security as a Lattice

Low-security information *may* flow to high-security contexts

Zdancewic. “Programming Languages for Information Security”
Security as a Lattice

High-security information may not flow to low-security contexts

Zdancewic. “Programming Languages for Information Security”
Security Typing

Int → Int

Simple Types
Security Typing

Security-Indexed Types

(\text{Int}^H \rightarrow \text{Int}^L \rightarrow ^H \text{Int})
Security Typing

\[ \text{Int}_L \lll \text{Int}_H \]

\[ \text{Int}_H \longrightarrow_L \text{Int}_L \lll \text{Int}_L \longrightarrow_H \text{Int}_H \]

Natural Subtyping Structure
let age : Int = 31
let salary : Int = 58000
let intToString : Int → String = ...
let print : String → Unit = ...
print(intToString(salary))
Scenario

let age : Int = 31
let salary : Int = 58000
let intToString : Int → String = ...
let print : String → Unit = ...
print(intToString(salary))

Type Error!

Security Typing
More Subtleties

High-security information may not flow to low-security contexts

Zdancewic. “Programming Languages for Information Security”
More Subtleties

High-security information may not flow to low-security contexts.

Zdancewic. “Programming Languages for Information Security”
Implicit Information Flows

fun b : BoolH =>
    let tt : BoolL = true
    let ff : BoolL = false
    if b then tt else ff

High-Security data can affect control flow of a program
Implicit Information Flows

fun b : BoolH =>
    let tt : BoolL = true
    let ff : BoolL = false
    if b then tt else ff

High-Security data can affect control flow of a program

What’s it’s Type?
Implicit Information Flows

fun b : BoolH =>
  let tt : BoolL = true
  let ff : BoolL = false
  if b then tt else ff

High-Security data can affect control flow of a program
Assignment Can Leak Info!

```ocaml
let r : BoolL ref = ref tt
fun b : BoolH =>
  if b then ()L else (r := ff; ()L)
```

High-Security information can escape via mutable state
Assignment Can Leak Info!

```ocaml
let r : BoolL ref = ref tt
fun b : BoolH =>
  if b then ()L else (r := ff; ()L)
```

High-Security information can escape via mutable state
Security Typing Judgment

$$\Gamma; \Sigma; \ell \vdash t : T$$
Security Typing Judgment

$$\Gamma; \Sigma; \ell \vdash t : T$$

How Can My Local Variables Behave?
Security Typing Judgment

\[ \Gamma; \Sigma; \ell \vdash t : T \]

- How Can My Local Variables Behave?
- How Can Mutable References Behave?
Security Typing Judgment

\[ \Gamma; \Sigma; \ell \vdash t : T \]

- How Can My Local Variables Behave?
- How Can Mutable References Behave?
- What Security Information can leak through Assignment

What Security Information can leak through Assignment
Security Typing Judgment

Γ; Σ; ℓ ⊢ t : T

How Does t Behave?
let age : IntL = 31L
let salary : IntH = 58000H
let intToString : IntL →L StringL = ...
let print : StringL →L UnitL = ...
print(intToString(salary))

Type Error!

Security Typing
Secure All the Things!

let age = 31;
let salary = 58000;
let intString = String;
let print = Unit;
print(intToString(salary))

Type Error!

Security Typing
Gradual Typing

Dynamic Typing

Simple Typing
Gradual Typing!

Simple Typing

Security Typing

Disney and Flanagan. “Gradual Information Flow Typing”
Gradual Typing

Fennell and Thiemann, Gradual Security Typing with References
let age : Int = 31
let salary : Int = 58000
let intToString : Int \rightarrow String = ...
let print : String \rightarrow Unit = ...
print(intToString(salary))
“Gradually Secure” Program

let age : Int = 31
let salary : Int = 58000
let intToString : Int → String = ...
let print : String → Unit = ...
print(intToString(salary))

Runtime Error!
let age : Int = 31
let salary : Int = 58000
let intToString : Int → String = ...
let print : String → Unit = ...
print(intToString(salary))
let age : Int = 31
let salary : Int = 58000
let intToString : Int -> String = ...
let print : String -> Unit = ...
print(intToString(age))

All Good!
What do types tell us?

```ocaml
let mix : IntL -> IntH -> IntL =
  fun pub priv =>
    ...
```

Local Reasoning Principles???
What do types tell us?

let mix : IntL -> IntH -> IntL =
  fun pub priv =>
    ...

Take 1: Upper-bounds on security tags

Constrains any individual run of the code

Weak security guarantee

Proof Technique: Wright-Felleisen Type Safety

Disney and Flanagan. “Gradual Information Flow Typing”

Fennell and Thiemann, Gradual Security Typing with References
What do types tell us?

let mix : IntL -> IntH -> IntL =
  fun pub priv =>
    ...

Take 2: Non-interference

Constrains relationship among runs of the code

Strong security guarantee

Proof Technique: Logical relations

Modular, compositional, static reasoning about security

Heintze and Riecke. The Slam Calculus: Programming with Secrecy and Integrity
What do types tell us?

```
let mix : IntL -> IntH -> IntL = ...
```

Constrains relationship among runs of the code.

Proof Technique: Logical relations

Modular, compositional, static reasoning about security

Heintze and Riecke. The Slam Calculus: Programming with Secrecy and Integrity

Milner Award Lecture: The Type Soundness Theorem That You Really Want to Prove (and Now You Can) - POPL 2018

Type systems—and the associated concept of “type...”

What do types tell us?

let mix : IntL → IntH → IntL =
  fun pub priv =>
    ...

Take 2: Non-interference
MISSION: achieve this richer meaning
Strong security guarantee

Modular, compositional, \textit{gradual} reasoning about security
Gradual Security

\[ \ell \in \text{Label} \]

\[ g \in \text{GLabel} ::= \ell \mid ? \]

\[ \text{Label} \subseteq \text{GLabel} \]
let age : Int = 31
let salary : Int = 58000
let intToString : Int → String = ...
let print : String → Unit = ...
print(intToString(age))
“Gradually Secure” Program

Desugared

let age : Int? = 31?
let salary : IntH = 58000?
let intToString : IntL ->? String? = ...
let print : StringL ->? Unit? = ...
print(intToString(age))

Gradual Language Embeds
Simply Typed and Security Typed Languages
Gradual Label Precision

\[ g_1 \sqsubseteq g_2 \]
Label Ordering

\[ l_1 \preceq l_2 \]
Consistent Label Ordering

\[ g_1 \preceq g_2 \]

\[ \ell_1 \preceq \ell_2 \]

for some \( \ell_1, \ell_2 \)
Consistent Ordering

Conservatively Extends Label Ordering
Consistent “Ordering”

$L \sim H$

$H \not\sim L$

$L \sim L$

$L \sim L$

$L \sim ?$

$L \sim ?$

Not really an order
Gradual Types

\[ U \in \text{GType} \]

\[ \text{Bool}_L \]

\[ \text{Int}_H \]

\[ \text{Bool}_? \]

\[ \text{TYPE} \subseteq \text{GType} \]

Just add gradual labels!
Gradual Types

$U_1 \subseteq U_2$

Covariant on function types!
Gradual Types

Consistent Subtyping

Conservatively extends subtyping (but not really a subtyping relation)

\[ U_1 \sim U_2 \]

if and only if

\[ T_1 <: T_2 \]

for some \( T_1, T_2 \)
Consistent Subtyping

\[ \text{Int}_L \preceq \text{Int}_H \]
\[ \text{Int}_L \npreceq \text{Bool}_H \]
\[ \text{Int}_H \npreceq \text{Int}_L \]
\[ \text{Int}_H \preceq \text{Int}_? \]
\[ \text{Int}_? \preceq \text{Int}_L \]
\[ \text{Int}_? \npreceq \text{Bool}_H \]

Conservatively Extends Subtyping
Consistent Subtyping

\[ \text{Int}_L \preceq \text{Int}_H \]

\[ \text{Int}_L \not\preceq \text{Bool}_H \]

\[ \text{Int}_H \not\preceq \text{Int}_L \]

\[ \text{Int}_H \preceq \text{Int}_? \]

\[ \text{Int}_? \preceq \text{Int}_L \]

\[ \text{Int}_? \not\preceq \text{Bool}_H \]

Not Transitive!
Consistent Subtyping

\( \text{Int}_L \preceq \text{Int}_H \)

\( \text{Int}_L \not\preceq \text{Bool}_H \)

\( \text{Int}_H \not\preceq \text{Int}_L \)

\( \text{Int}_H \preceq \text{Int}? \)

\( \text{Int}? \preceq \text{Int}_L \)

\( \text{Int}? \not\preceq \text{Bool}_H \)

Does **NOT** denote safe substitutibility
Consistent Subtyping

\[ \text{Int}_L \preceq \text{Int}_H \]
\[ \text{Int}_L \not\prec \text{Bool}_H \]
\[ \text{Int}_H \not\prec \text{Int}_L \]
\[ \text{Int}_H \preceq \text{Int}_? \]
\[ \text{Int}_? \preceq \text{Int}_L \]
\[ \text{Int}_? \not\prec \text{Bool}_H \]

Does NOT denote safe substitutability

Not a Subtyping Relation!

Not Transitive!
Lifting Typing Rules

\[ \Gamma; \Sigma; \ell_c \vdash t_1 : T_{11} \lll_{\ell_x} T_{12} \]
\[ T_2 \lll T_{11} \]
\[ \Gamma; \Sigma; \ell_c \vdash t_1 \ t_2 : T_{12} \lll \ell_x \]

\[ \Gamma; \Sigma; g_c \vdash \tilde{t}_1 : U_{11} \lll_{g_x} U_{12} \]
\[ U_2 \lll U_{11} \]
\[ \Gamma; \Sigma; g_c \vdash \tilde{t}_1 \ \tilde{t}_2 : U_{12} \lll g_x \]

\[ \Gamma; \Sigma; g_c \vdash \tilde{t}_1 \ \tilde{t}_2 : U_{12} \lll g_x \]

\[ \Gamma; \Sigma; g_c \vdash \tilde{t}_1 : U_{11} \lll_{g_x} U_{12} \]
\[ g_c \lll g_x \lll g' \]
\[ \Gamma; \Sigma; g_c \vdash \tilde{t}_1 \ \tilde{t}_2 : U_{12} \lll g_x \]
Dynamic Semantics: Runtime Type Safety Argument

Garcia et al. Abstracting Gradual Typing (POPL 2016)
Noninterference (roughly)

\[ \Gamma; \Sigma; g \vdash t : U \implies \Gamma; \Sigma; g \models t : U \]

Syntactic Type Judgment

Semantic Type Judgment

Semantic Type Soundness
Theses

• Types let you reason about program fragments

• Type Systems are not their Type Checkers
  • Type Systems are for reasoning
  • Type Checkers are for enforcement
  • Dynamic Checks are for enforcement too!
Conclusion

• Gradual typing is relative: not just for “scripting”

• Gradual typing conservatively extends two related languages

  • Syntax

  • Dynamic Semantics

  • Semantics of types
Image Credits

- “see no evil…” by ucumari (flickr)
- “concentration” by cdell (flickr)
- “Can you say ‘SAWHEEET’?” by locomotion (flickr)