Gradual Information Flow Control

HCSS 2016

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Luminous Fennell, *Peter Thiemann*

2016-05-11

Assist programmers in guaranteeing secure information flow

Goal of our research

Assist programmers in guaranteeing secure information flow

1. when designing new systems
2. when extending legacy systems
3. when fixing insecure systems

What is secure information flow?

- Assume a lattice of *levels of confidentiality* with `LOW` $\leq$ `HIGH`

What is secure information flow?

- Assume a lattice of *levels of confidentiality* with `LOW` $\leq$ `HIGH`
- For example, `LOW` $\approx$ public and `HIGH` $\approx$ secret

What is secure information flow?

- Assume a lattice of *levels of confidentiality* with `LOW` $\leq$ `HIGH`
- For example, `LOW` $\approx$ public and `HIGH` $\approx$ secret
- If `leak` is a publicly readable variable and `hi` contains a secret value, then . . .

What is secure information flow?

- Assume a lattice of *levels of confidentiality* with $\texttt{LOW} \leq \texttt{HIGH}$
- For example, $\texttt{LOW} \approx$ public and $\texttt{HIGH} \approx$ secret
- If `leak` is a publicly readable variable and `hi` contains a secret value, then ...
  - an example for an *explicit flow* is a direct assignment

```
leak = hi;
```

What is secure information flow?

- Assume a lattice of *levels of confidentiality* with $\texttt{LOW} \leq \texttt{HIGH}$
- For example, $\texttt{LOW} \approx$ public and $\texttt{HIGH} \approx$ secret
- If `leak` is a publicly readable variable and `hi` contains a secret value, then ...
  - an example for an *explicit flow* is a direct assignment

    ```
    leak = hi;
    ```

  - an example for an *implicit flow* is a conditional assignment

    ```
    leak = 0;
    if (hi) { leak = 1; }
    ```

What is secure information flow?

- Assume a lattice of *levels of confidentiality* with `LOW` $\leq$ `HIGH`
- For example, `LOW` $\approx$ public and `HIGH` $\approx$ secret
- If `leak` is a publicly readable variable and `hi` contains a secret value, then ...
  - an example for an *explicit flow* is a direct assignment

    ```
    leak = hi;
    ```

  - an example for an *implicit flow* is a conditional assignment

    ```
    leak = 0;
    if (hi) { leak = 1; }
    ```

- **Secure information flow control rules out explicit and implicit flows of secret values to public outputs**

Approaches to information flow control (IFC)

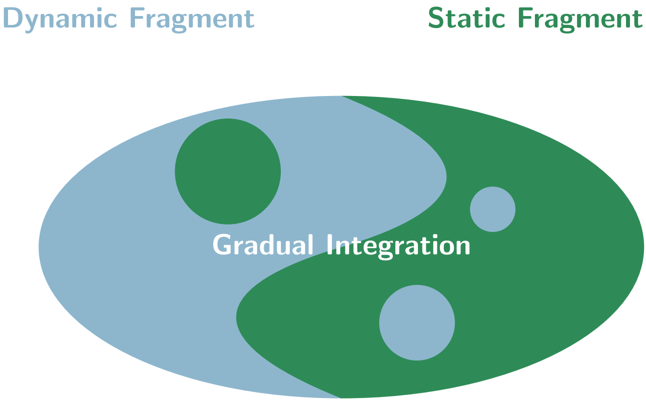
Dynamic

- ⊕ Flexible
- ⊖ Fragile
- ⊖ Slow

Static

- ⊖ Conservative
- ⊕ Robust
- ⊕ Fast

Approaches to information flow control (IFC)



Our proposal: gradual information flow

Combination of dynamic and static information flow control

- STOP2011: Gradual Information Flow (Disney & Flanagan)
- CSF2013: Gradual Security with References (F&T)
- ESOP2014: Gradual Annotated Typing (F&T)
- ECOOP2016: LJGS—Gradual Security for Java (F&T)

LJGS — Gradual Information Flow for Java

Goal: make IFC readily accessible to Java programmers

1. Gradual integration of dynamic and static IFC
   built on top of the standard Java type system

LJGS — Gradual Information Flow for Java

Goal: make IFC readily accessible to Java programmers

1. Gradual integration of dynamic and static IFC
   built on top of the standard Java type system
2. Polymorphic security signatures for methods

LJGS — Gradual Information Flow for Java

Goal: make IFC readily accessible to Java programmers

1. Gradual integration of dynamic and static IFC built on top of the standard Java type system
2. Polymorphic security signatures for methods
3. Flow-sensitive treatment of local variables

LJGS — Gradual Information Flow for Java

Goal: make IFC readily accessible to Java programmers

1. Gradual integration of dynamic and static IFC
   built on top of the standard Java type system
2. Polymorphic security signatures for methods
3. Flow-sensitive treatment of local variables
4. No run-time overhead for statically checked code
   $\Rightarrow$ only **dynamic values** carry run-time labels

LJGS — Gradual Information Flow for Java

Goal: make IFC readily accessible to Java programmers

1. Gradual integration of dynamic and static IFC
   built on top of the standard Java type system
2. Polymorphic security signatures for methods
3. Flow-sensitive treatment of local variables
4. No run-time overhead for statically checked code
   $\Rightarrow$ only **dynamic values** carry run-time labels

Added value of LJGS: combination of features ?? – ??

Example LJGS Class

```
class Example1 {
  String<LOW> low;
  // Field annotated with fixed security level

  int maxWithMessage(int x, int y)
    where { @x ≤ @return, @y ≤ @return}
    // Parameter constraints: @return subsumes @x and @y
    effect { LOW }
    // Write effects to check implicit flows
  {
    if (x ≤ y) { x = y; }
    this.low = "max was called";
    return x;
  }
}
```

Signature example LJGS class

```
class Example2 {
  String<HIGH> high; String<LOW> low;
  String<⋆> dyn;
  // Field annotated as dynamic; flow sensitive

  void updateSecurely (String s, boolean isPublic)
    where {@s ≤ ⋆, @isPublic ≤ LOW }
      effect { LOW , ⋆ } {
    this.dyn = s;
    this.high = (HIGH ⇐ ⋆) this.dyn;
    // Cast to mediate between dynamic and static types
    if (isPublic) {
      this.low = (LOW ⇐ ⋆) this.dyn;
    }
  }
}
```

LJGS Typing

- Typing generates constraints of the form
  - $\{\texttt{LOW} \leq \alpha\}$ some fixed level as lower bound
  - $\{\alpha \leq \texttt{HIGH}\}$ ... or upper bound
  - $\{\alpha_1 \leq \alpha_2\}$ flow
- Type variables $\alpha, \alpha_1, \alpha_2 \dots$
- Context variable γ (confidentiality of enclosing conditionals)
- Local variables are bound to different type variables at each program point (flow sensitivity) via typing environment Γ
- Constraints collected in constraint set $\mathcal{C}$
- Global **write effects** $\mathcal{E}$ (a set of security types)

Flow Sensitivity and Indirect Flows

```
                        // Γ_0 = { }
x = this.lowField;      // Γ_1 = [x ↦ α_1]
x = this.highField;     // Γ_2 = [x ↦ α_2]
```

$$\mathcal{C} = \{\texttt{LOW} \leq \alpha_1, \texttt{HIGH} \leq \alpha_2, \gamma \leq \alpha_1, \gamma \leq \alpha_2\} \ldots$$

Flow Sensitivity and Indirect Flows

```
                    // Γ0 = { }
x = this.lowField;  // Γ1 = [x ↦ α1]
x = this.highField; // Γ2 = [x ↦ α2]
```

$$\mathcal{C} = \{\texttt{LOW} \leq \alpha_1, \texttt{HIGH} \leq \alpha_2, \gamma \leq \alpha_1, \gamma \leq \alpha_2\} \ldots$$

```
                                 // Γ1 = [y ↦ α]
if (y) { x = this.lowField; }    // Γ2 = [y ↦ α, x ↦ β]
```

$$\mathcal{C} = \{\alpha \leq \gamma, \texttt{LOW} \leq \beta, \gamma \leq \beta\} \ldots$$

Interpretation of Constraints

Easy for static security levels:

- $\mathcal{C} = \{\alpha \leq \texttt{LOW}, \texttt{HIGH} \leq \beta, \alpha \leq \beta\} \dots$
- Find a mapping from $\alpha, \beta, \dots$ to elements of the security lattice that does not violate the constraints, i.e., such that $\texttt{HIGH} \not\leq \texttt{LOW}$.

But how to include "Type Dynamic"?

Interpretation of Constraints

Easy for static security levels:

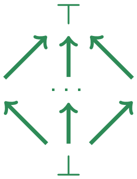
- $\mathcal{C} = \{\alpha \leq \texttt{LOW}, \texttt{HIGH} \leq \beta, \alpha \leq \beta\} \ldots$
- Find a mapping from $\alpha, \beta, \ldots$ to elements of the security lattice that does not violate the constraints, i.e., such that $\texttt{HIGH} \not\leq \texttt{LOW}$.

But how to include "Type Dynamic"?

Naive Approach

- $\top \leq \star$
- `this.dynField = this.statField` is allowed
- `if(this.statField) this.dynField = 42;` is allowed
- Consequence:
  - no clear separation of the static/dynamic fragments
  - run-time checks with static types "all over the place"

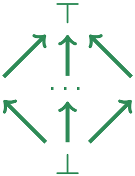
Including "Type Dynamic"



static information

Including “Type Dynamic”

“dynamic”
$\star$



dynamic information **static information**

Type Dynamic for Values

Recall the max function

```
int max(int x, int y)
  where { @x ≤ @return, @y ≤ @return } {
  if (x ≤ y) {
    x = y;
  }
  return x;
}
```

Type Dynamic for Values

```
int max(int x, int y)
  where { @x ≤ @return, @y ≤ @return } { ... }


int d1[⋆]; int d2[⋆]; int sH[HIGH]; int sL[LOW];

void callingMax() where { } effect { ⋆, HIGH } {
  this.d1 = max(this.d1, this.d2);    // ok
  this.sH = max(this.sH, 42);    // ok




}
```

Type Dynamic for Values

```
int max(int x, int y)
  where { @x ≤ @return, @y ≤ @return } { ... }


int d1[⋆]; int d2[⋆]; int sH[HIGH]; int sL[LOW];

void callingMax() where { } effect { ⋆, LOW } {
  this.d1 = max(this.d1, this.d2);   // ok
  this.sH = max(this.sH, 42);   // ok
  this.d1 = max(this.d1, this.sH); // type error
   // ^ no solution for @return
  this.sL = this.d1                // type error
   // ^ ⋆ ≰ LOW
}
```

Type Dynamic for Values

```
int max(int x, int y)
  where { @x ≤ @return, @y ≤ @return } { ... }


int d1[⋆]; int d2[⋆]; int sH[HIGH]; int sL[LOW];

void callingMax() where { } effect { ⋆, LOW } {
  this.d1 = max(this.d1, this.d2);    // ok
  this.sH = max(this.sH, 42);    // ok
  this.d1 = max(this.d1, (⋆ ⇚ HIGH) this.sH); // ok
   // ^ value cast to dynamic
  this.sL = (LOW ⇚ ⋆) this.d1       // run-time error
   // ^ failing value cast from dynamic
}
```

Type Dynamic for Values

```
int max(int x, int y)
  where { @x ≤ @return, @y ≤ @return } { ... }


int d1[⋆]; int d2[⋆]; int sH[HIGH]; int sL[LOW];

void callingMax() where { } effect { ⋆, LOW } {
  this.d1 = max(this.d1, this.d2);    // ok
  this.sH = max(this.sH, 42);    // ok
  this.d1 = max(this.d1, (⋆ ⇐ HIGH) this.sH); // ok
   // ^ value cast to dynamic
  this.sL = (LOW ⇐ ⋆) this.d1       // run-time error
   // ^ failing value cast from dynamic
}
```

Value casts

- Declare security levels to be represented at run-time
- Check statically unknown security levels

Type Dynamic for Contexts

```
int d1[⋆]; int d2[⋆]; int sH[HIGH];

void updates() where { } and { ⋆, HIGH } {
  if (this.d1 == 42) {
    this.d2 = this.d1; // ok


  }
  if (this.sH == 42) {
    this.sH += 1; // ok


  }
}
```

Type Dynamic for Contexts

```
int d1[⋆]; int d2[⋆]; int sH[HIGH];

void updates() where { } and { ⋆, HIGH } {
  if (this.d1 == 42) {
    this.d2 = this.d1; // ok
    this.sH = 42 // type error
    // ^ static update in dynamic context
  }
  if (this.sH == 42) {
    this.sH += 1; // ok
    this.d1 = this.d2 // type error
    // ^ dynamic update in static context
  }
}
```

Type Dynamic for Contexts

```
int d1[⋆]; int d2[⋆]; int sH[HIGH];

void updates() where { } and { ⋆, HIGH } {
  if (this.d1 == 42) {
    this.d2 = this.d1; // ok
    (⋆ ⇛ HIGH) {this.sH = 42;} // ok

  }
  if (this.sH == 42) {
    this.sH += 1; // ok
    (HIGH ⇛ ⋆) {this.d1 = this.d2;} // ok

  }
}
```

Type Dynamic for Contexts

```
int d1[⋆]; int d2[⋆]; int sH[HIGH];

void updates() where { } and { ⋆, HIGH } {
  if (this.d1 == 42) {
    this.d2 = this.d1; // ok
    (⋆ ⇛ HIGH) {this.sH = 42;} // ok

  }
  if (this.sH == 42) {
    this.sH += 1; // ok
    (HIGH ⇛ ⋆) {this.d1 = this.d2;} // ok

  }
}
```

- Static updates cannot be checked in dynamic contexts
- Dynamic updates need the context represented at run-time

Public Contexts

```
int d1[⋆]; int d2[⋆]; int sH[HIGH];

void updates2() where { } and { ⋆, HIGH } {
  this.d2 = this.d1; // dynamic context check
  this.sH = 42       // static context check
                     // type error ???
}
```

Public Contexts

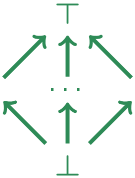
```
int d1[⋆]; int d2[⋆]; int sH[HIGH];

void updates2() where { } and { ⋆, HIGH } {
  this.d2 = this.d1; // dynamic context check
  this.sH = 42       // static context check
                     // type error ???
}
```

Need a *public context* that can accept both, static and dynamic updates.

Including “Type Dynamic”

Greatest lower bound of $\{\star, \texttt{HIGH}\}$?

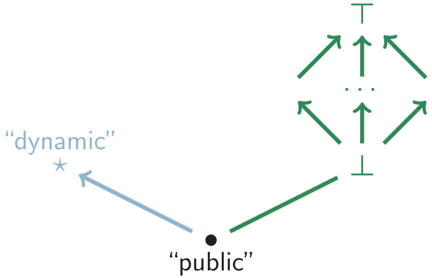
“dynamic”
$\star$



dynamic information **static information**

Including "Type Dynamic"

Greatest lower bound of $\{\star, \texttt{HIGH}\}$?



dynamic information **static information**

Public Contexts

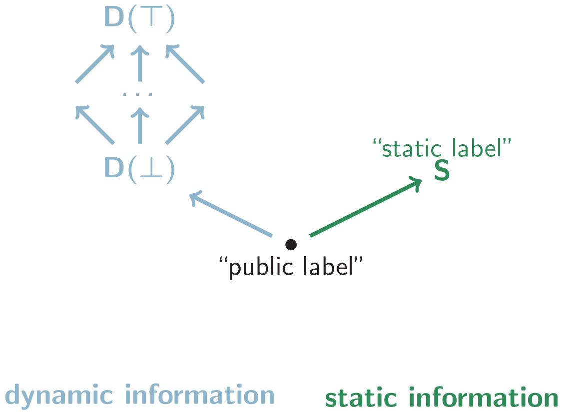
```
int d1[⋆]; int d2[⋆]; int sH[HIGH];

void updates2() where { } and { • } {
  this.d2 = this.d1; // ok
  this.sH = 42       // ok

}
```

Solution

A *public context* of type • can accept both, static and dynamic updates.

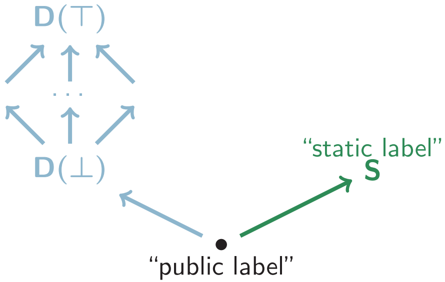
- . . . as it is trivially secure
- calling `update2()` at the top-level is fine
- `if(sH == 42){ update2(); }` is a type error

In the LJGS core calculus, all values carry run-time labels. They are the mirror image to security types:



Run-Time Labels

In the LJGS core calculus, all values carry run-time labels. They are the mirror image to security types:



dynamic information **static information**

static labels carry no information and may be erased

Dynamic IFC

1. NSU (no sensitive upgrade; Zdancewic, Austin&Flanagan)
2. PU (permissive upgrade; Austin&Flanagan)
3. FE (faceted execution; Austin&Flanagan)
4. Hybrid extension relying on side effect analysis

Implementation

Combination of NSU and side effect analysis

Main application areas

Securing legacy code

- Incorporate legacy code by treating it as dynamic
- Code needs to be recompiled or available for bytecode transformation

Code with explicit security management

- Confidentiality of a data item depends on a condition on public data
- Static check on in the scope of a static analysis
- Insert casts where needed

Conclusions

UNI FREIBURG

- LJGS integrates static and dynamic IFC by gradual typing
- Static and dynamic code fragments interact through casts
- No run-time overhead for purely static fragments
- Methodology applicable to other areas (e.g., units of measure)
- More details in the ECOOP 2016 paper/techreport
  - Meta-theoretic results (NI, type safety)
  - Language construct to inspect dynamic labels

Status

- Type checker available (github)
- Implementation (bytecode) close to completion
- To do: exceptions