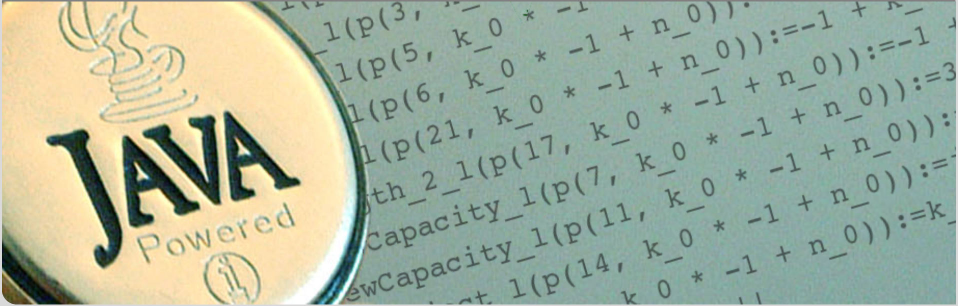
Karlsruhe Institute of Technology

Applications of Formal Verification

Deductive Verification of Information Flow Properties of Java Programs

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov | SS 2012

KIT – INSTITUT FÜR THEORETISCHE INFORMATIK



1. Non-Interference
   - Definition
   - Reformulation and Formalization – Alternating Quantifiers
   - Reformulation and Formalization – Self-Composition

2. Declassification

3. Termination-sensitive Non-interference

Non-Interference

Prominent information flow property: **non-interference**

Simple case:

- deterministic, terminating, imperative program *P*
- program variables of *P* are partitioned in
  - low-security variables *low* and
  - high-security variables *high*
- In the following, non-interference means *high* do not interfere with *low* in *P* (=no information flows from *high* to *low*)

Definition (Non-interference – not quite formal)

When starting *P* with arbitrary values for *low*, the values of *low* after executing *P* are independent of the choices of *high*.

Non-Interference

Karlsruhe Institute of Technology

Prominent information flow property: **non-interference**

Simple case:

- deterministic, terminating, imperative program *P*
- program variables of *P* are partitioned in
  - low-security variables *low* and
  - high-security variables *high*
- In the following, non-interference means *high* do not interfere with *low* in *P* (=no information flows from *high* to *low*)

Definition (Non-interference – not quite formal)

When starting *P* with arbitrary values for *low*, the values of *low* after executing *P* are independent of the choices of *high*.

Non-Interference

Karlsruhe Institute of Technology

Prominent information flow property: **non-interference**

Simple case:

- deterministic, terminating, imperative program *P*
- program variables of *P* are partitioned in
  - low-security variables *low* and
  - high-security variables *high*
- In the following, non-interference means *high* do not interfere with *low* in *P* (=no information flows from *high* to *low*)

Definition (Non-interference – not quite formal)

When starting *P* with arbitrary values for *low*, the values of *low* after executing *P* are independent of the choices of *high*.

Non-Interference

Prominent information flow property: **non-interference**

Simple case:

- deterministic, terminating, imperative program *P*
- program variables of *P* are partitioned in
  - low-security variables *low* and
  - high-security variables *high*
- In the following, non-interference means *high* do not interfere with *low* in *P* (=no information flows from *high* to *low*)

Definition (Non-interference – not quite formal)

When starting *P* with arbitrary values for *low*, the values of *low* after executing *P* are independent of the choices of *high*.

Non-Interference

Karlsruhe Institute of Technology

Prominent information flow property: **non-interference**

Simple case:

- deterministic, terminating, imperative program *P*
- program variables of *P* are partitioned in
  - low-security variables *low* and
  - high-security variables *high*
- In the following, non-interference means *high* do not interfere with *low* in *P* (=no information flows from *high* to *low*)

Definition (Non-interference – not quite formal)

When starting *P* with arbitrary values for *low*, the values of *low* after executing *P* are independent of the choices of *high*.

Non-Interference

Prominent information flow property: **non-interference**

Simple case:

- deterministic, terminating, imperative program *P*
- program variables of *P* are partitioned in
  - low-security variables *low* and
  - high-security variables *high*
- In the following, non-interference means *high* do not interfere with *low* in *P* (=no information flows from *high* to *low*)

Definition (Non-interference – not quite formal)

When starting *P* with arbitrary values for *low*, the values of *low* after executing *P* are independent of the choices of *high*.

Examples

Which methods are secure?

```
class MiniExamples {
  public int l;
  private int h;

  void m_1() {
    l = h;
  }

  void m_2() {
    if (l>0) {h=1;}
    else {h=2;};
  }

  void m_3() {
    if (h>0) {l=1;}
    else {l=2;};
  }

  void m_4() {
    h=0; l=h;
  }
```

Examples

Which methods are secure?

```
class MiniExamples {
  public int l;
  private int h;

  void m_1() {
    l = h;
  }

  void m_2() {
    if (l>0) {h=1;}
    else {h=2;};
  }

  void m_3() {
    if (h>0) {l=1;}
    else {l=2;};
  }

  void m_4() {
    h=0; l=h;
  }
```

Examples

Karlsruhe Institute of Technology

Which methods are secure?

```
class MiniExamples {
  public int l;
  private int h;

  void m_1() {
    l = h;
  }

  void m_2() {
    if (l>0) {h=1;}
    else {h=2;};
  }

  void m_3() {
    if (h>0) {l=1;}
    else {l=2;};
  }

  void m_4() {
    h=0; l=h;
  }
```

Examples

Which methods are secure?

```
class MiniExamples {
  public int l;
  private int h;

  void m_1() {
    l = h;
  }

  void m_2() {
    if (l>0) {h=1;}
    else {h=2;};
  }

  void m_3() {
    if (h>0) {l=1;}
    else {l=2;};
  }

  void m_4() {
    h=0; l=h;
  }
```

Examples

Which methods are secure?

```
class MiniExamples {
  public int l;
  private int h;

  void m_1() {
    l = h;
  }

  void m_2() {
    if (l>0) {h=1;}
    else {h=2;};
  }

  void m_3() {
    if (h>0) {l=1;}
    else {l=2;};
  }

  void m_4() {
    h=0; l=h;
  }
```

Examples

Which methods are secure?

```
void m_5() {
  l=h; l=l-h;
}
```

```
void m_6() {
  if (false) l=h;
}
```

```
}
```

Examples

Which methods are secure?

```
void m_5() {
  l=h; l=l-h;
}
```

```
void m_6() {
  if (false) l=h;
}
```

```
}
```

Examples

Which methods are secure?

```
void m_5() {
  l=h; l=l-h;
}
```

```
void m_6() {
  if (false) l=h;
}
```

}

Non-Interference

Definition (Low-equivalence on states)

Two states are low-equivalent if they assign the same values to low variables.

Definition (Non-interference)

Starting P in two arbitrary low-equivalent states results in two final states that are also low-equivalent.

Non-Interference

Karlsruhe Institute of Technology

Definition (Low-equivalence on states)

Two states are low-equivalent if they assign the same values to low variables.

Definition (Non-interference)

Starting P in two arbitrary low-equivalent states results in two final states that are also low-equivalent.

Non-Interference in JavaDL – Alternating Quantifiers

Non-interference encoding in JavaDL (v1)

For all low input values in_l, there exist low output values r such that for all high input values in_h, if we assign the values in_l to the program variables *low* and in_h to the program variables *high*, then after execution of P the values of *low* are r.

$$\forall in_l \exists r \forall in_h (\{low := in_l \parallel high := in_h\}[P]low = r)$$

- **Problem:** not suitable for automatic verification $\rightsquigarrow$ instantiation of existential quantifier difficult.

Non-Interference in JavaDL – Alternating Quantifiers

Non-interference encoding in JavaDL (v1)

For all low input values in_l, there exist low output values r such that for all high input values in_h, if we assign the values in_l to the program variables *low* and in_h to the program variables *high*, then after execution of P the values of *low* are r.

$$\forall in_l \exists r \forall in_h (\{low := in_l \mid\mid high := in_h\}[P]low = r)$$

- **Problem:** not suitable for automatic verification $\rightsquigarrow$ instantiation of existential quantifier difficult.

Non-Interference in JavaDL – Alternating Quantifiers

Non-interference encoding in JavaDL (v1)

For all low input values in_l, there exist low output values r such that for all high input values in_h, if we assign the values in_l to the program variables *low* and in_h to the program variables *high*, then after execution of P the values of *low* are r.

$$\forall in_l \exists r \forall in_h (\{low := in_l \mid\mid high := in_h\}[P]low = r)$$

- **Problem:** not suitable for automatic verification $\rightsquigarrow$ instantiation of existential quantifier difficult.

Non-Interference in JavaDL – Alternating Quantifiers

Non-interference encoding in JavaDL (v1)

For all low input values in_l, there exist low output values r such that for all high input values in_h, if we assign the values in_l to the program variables *low* and in_h to the program variables *high*, then after execution of P the values of *low* are r.

$$\forall in_l \exists r \forall in_h (\{low := in_l \mid\mid high := in_h\}[P]low = r)$$

- **Problem:** not suitable for automatic verification $\rightsquigarrow$ instantiation of existential quantifier difficult.

Non-Interference in JavaDL – Alternating Quantifiers

Non-interference encoding in JavaDL (v1)

For all low input values in_l, there exist low output values r such that for all high input values in_h, if we assign the values in_l to the program variables *low* and in_h to the program variables *high*, then after execution of P the values of *low* are r.

$$\forall in_l \exists r \forall in_h (\{low := in_l \,||\, high := in_h\}[P]low = r)$$

- **Problem:** not suitable for automatic verification $\rightsquigarrow$ instantiation of existential quantifier difficult.

Non-Interference in JavaDL – Alternating Quantifiers

Non-interference encoding in JavaDL (v1)

For all low input values in_l, there exist low output values r such that for all high input values in_h, if we assign the values in_l to the program variables *low* and in_h to the program variables *high*, then after execution of P the values of *low* are r.

$$\forall in_l \exists r \forall in_h (\{low := in_l \mid\mid high := in_h\}[P] low = r)$$

- **Problem:** not suitable for automatic verification $\rightsquigarrow$ instantiation of existential quantifier difficult.

Non-Interference in JavaDL – Alternating Quantifiers

Non-interference encoding in JavaDL (v1)

For all low input values in_l, there exist low output values r such that for all high input values in_h, if we assign the values in_l to the program variables *low* and in_h to the program variables *high*, then after execution of P the values of *low* are r.

$$\forall in_l \exists r \forall in_h (\{low := in_l \mid\mid high := in_h\}[P]low = r)$$

- **Problem:** not suitable for automatic verification $\rightsquigarrow$ instantiation of existential quantifier difficult.

Non-Interference in JavaDL – Self-Composition

Non-interference encoding in JavaDL (v2)

Running two instances of *P* on the same low values but on arbitrary high values results in low variables which have the same values.

$$
\begin{array}{l}
\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2 \; \{low := in_l\}(\\
\qquad \{high := in_h^1\}[P]out_l^1 = low \\
\qquad \land \{high := in_h^2\}[P]out_l^2 = low \\
\qquad \to out_l^1 = out_l^2 \\
)
\end{array}
$$

Non-Interference in JavaDL – Self-Composition

Non-interference encoding in JavaDL (v2)

Running two instances of P on the same low values but on arbitrary high values results in low variables which have the same values.

$$
\begin{aligned}
&\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2\ \{low := in_l\}(\\
&\qquad \{high := in_h^1\}[P]out_l^1 = low \\
&\qquad \wedge \{high := in_h^2\}[P]out_l^2 = low \\
&\qquad \rightarrow out_l^1 = out_l^2 \\
&)
\end{aligned}
$$

Non-Interference in JavaDL – Self-Composition

Non-interference encoding in JavaDL (v2)

Running two instances of *P* on the same low values but on arbitrary high values results in low variables which have the same values.

$$
\begin{array}{l}
\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2 \; \{low := in_l\}(\\
\qquad \{high := in_h^1\}[P]out_l^1 = low \\
\quad \wedge \{high := in_h^2\}[P]out_l^2 = low \\
\quad \rightarrow out_l^1 = out_l^2 \\
)
\end{array}
$$

Non-Interference in JavaDL – Self-Composition

Non-interference encoding in JavaDL (v2)

Running two instances of P on the same low values but on arbitrary high values results in low variables which have the same values.

$$
\begin{aligned}
&\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2 \ \{low := in_l\}(\\
&\qquad \{high := in_h^1\}[P]out_l^1 = low \\
&\qquad \land \{high := in_h^2\}[P]out_l^2 = low \\
&\qquad \rightarrow out_l^1 = out_l^2 \\
&)
\end{aligned}
$$

Declassification

Let $T(high, low)$ be a term. Intuitively: The only thing the attacker is allowed to learn about the secret inputs is the value of T in the initial state.

Definition (Non-interference w/ declassification)

Starting P in two arbitrary low-equivalent states coinciding in the value of T results in two final states that are also low-equivalent.

Declassification

Let $T(high, low)$ be a term. Intuitively: The only thing the attacker is allowed to learn about the secret inputs is the value of T in the initial state.

Definition (Non-interference w/ declassification)

Starting P in two arbitrary low-equivalent states coinciding in the value of T results in two final states that are also low-equivalent.

Declassification in JavaDL – Self-Composition

Encoding non-interference w/ declassification in JavaDL

Running two instances of P on the same low values and arbitrary high values coinciding on T results in low variables which have the same values.

$$
\begin{aligned}
&\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2 \; \{low := in_l\}(\\
&\qquad \{high := in_h^1\}T = \{high := in_h^2\}T \\
&\qquad \wedge \{high := in_h^1\}[P]out_l^1 = low \\
&\qquad \wedge \{high := in_h^2\}[P]out_l^2 = low \\
&\qquad \rightarrow out_l^1 = out_l^2 \\
&)
\end{aligned}
$$

Declassification in JavaDL – Self-Composition

Encoding non-interference w/ declassification in JavaDL

Running two instances of P on the same low values and arbitrary high values coinciding on T results in low variables which have the same values.

$$
\begin{aligned}
&\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2 \ \{low := in_l\}(\\
&\qquad \{high := in_h^1\}T = \{high := in_h^2\}T \\
&\qquad \land \{high := in_h^1\}[P]out_l^1 = low \\
&\qquad \land \{high := in_h^2\}[P]out_l^2 = low \\
&\qquad \rightarrow out_l^1 = out_l^2 \\
&)
\end{aligned}
$$

Declassification in JavaDL – Alternating Quantifiers

Encoding non-interference w/ declassification in JavaDL

For all values of T, for all low input values in_l, there exist low output values r such that for all high input values in_h, if we assign the values in_l to the program variables *low* and in_h to the program variables *high*, then after execution of P the values of *low* are r.

$$\forall d \forall in_l \exists r \forall in_h \{low := in_l \mid\mid high := in_h\}(T = d \rightarrow [P]low = r)$$

Declassification in JavaDL – Alternating Quantifiers

Encoding non-interference w/ declassification in JavaDL

For all values of T, for all low input values in_l, there exist low output values r such that for all high input values in_h, if we assign the values in_l to the program variables *low* and in_h to the program variables *high*, then after execution of P the values of *low* are r.

$$\forall d \forall in_l \exists r \forall in_h \{low := in_l \mid\mid high := in_h\}(T = d \rightarrow [P]low = r)$$

Adding Termination-sensitivity

We retract the requirement that P must always terminate.

Definition (Termination-sensitive non-interference)

Starting P in two arbitrary low-equivalent states either results in two non-terminating runs or in two final states that are also low-equivalent.

$$
\begin{aligned}
&\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2 \ \{low := in_l\}(\\
&\quad \{high := in_h^1\}\langle P\rangle true \ \land \ \{high := in_h^2\}\langle P\rangle true \land \\
&\qquad (\{high := in_h^1\}\langle P\rangle out_l^1 = low \land \\
&\qquad \{high := in_h^2\}\langle P\rangle out_l^2 = low \rightarrow \\
&\qquad out_l^1 = out_l^2) \\
&) \ \lor \ (\{high := in_h^1\}[P]false \land \{high := in_h^2\}[P]false)
\end{aligned}
$$

Adding Termination-sensitivity

We retract the requirement that P must always terminate.

Definition (Termination-sensitive non-interference)

Starting P in two arbitrary low-equivalent states either results in two non-terminating runs or in two final states that are also low-equivalent.

$$
\begin{aligned}
&\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2 \; \{low := in_l\}(\\
&\quad \{high := in_h^1\}\langle P\rangle true \;\wedge\; \{high := in_h^2\}\langle P\rangle true \wedge \\
&\qquad (\{high := in_h^1\}\langle P\rangle out_l^1 = low \wedge \\
&\qquad\ \{high := in_h^2\}\langle P\rangle out_l^2 = low \rightarrow \\
&\qquad\ out_l^1 = out_l^2) \\
&) \;\vee\; (\{high := in_h^1\}[P]false \wedge \{high := in_h^2\}[P]false)
\end{aligned}
$$

Adding Termination-sensitivity

We retract the requirement that P must always terminate.

Definition (Termination-sensitive non-interference)

Starting P in two arbitrary low-equivalent states either results in two non-terminating runs or in two final states that are also low-equivalent.

$$
\begin{array}{l}
\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2 \; \{low := in_l\}(\\
\quad \{high := in_h^1\}\langle P\rangle true \; \wedge \; \{high := in_h^2\}\langle P\rangle true \wedge \\
\qquad (\{high := in_h^1\}\langle P\rangle out_l^1 = low \wedge \\
\qquad \;\{high := in_h^2\}\langle P\rangle out_l^2 = low \rightarrow \\
\qquad \; out_l^1 = out_l^2) \\
) \; \vee \; (\{high := in_h^1\}[P]false \wedge \{high := in_h^2\}[P]false)
\end{array}
$$

Adding Termination-sensitivity

Another encoding of termination-sensitive non-interf.

For every low input, if P terminates for some high input, then it teminates for all high inputs, and with the same low output.

$$
\begin{aligned}
&\forall in_l \; \{low := in_l\}(\\
&\qquad \exists in_h\{high := in_h\}\langle P\rangle true \rightarrow \\
&\qquad \exists r \forall in_h\{high := in_h\}\langle P\rangle low = r \\
&)
\end{aligned}
$$

Not Covered Here

- Concurrency / nondeterminism
- Objects & heap
- Properties beyond non-interference (e.g., data integrity)