Deductive Verification of Information Flow Properties of Java Programs

Christoph Scheben | July 13, 2011
Aim

Static verification of explicit and implicit flows in Java programs:

1. Program-level specification of information flow properties
   - considered programming language: Java
   - considered specification language: JML

2. Deductive verification of such properties without approximation of information flow dependencies
   - verification system: KeY
   - low level specification: JavaDL (Java Dynamic Logic)
Prominent information flow property: non-interference

Simple case:
- Program $P$
- Partition of the program variables of $P$ in low security variables $low$ and high security variables $high$

**Definition (Non-interference – Version 1)**

For program $P$ the high variables $high$ do not interfere with the low variables $low$

$\iff$

when starting $P$ with arbitrary values for $low$, then the values of $low$ after executing $P$, are independent of the choices of $high$. 
Non-Interference

Prominent information flow property: non-interference

Simple case:
- program $P$
- partition of the program variables of $P$ in
  - low security variables $\text{low}$ and
  - high security variables $\text{high}$

Definition (Non-interference – Version 1)
For program $P$ the high variables $\text{high}$ do not interfere with the low variables $\text{low}$

$\iff$

when starting $P$ with arbitrary values for $\text{low}$, then the values of $\text{low}$ after executing $P$, are independent of the choices of $\text{high}$. 
Non-Interference

Prominent information flow property: non-interference

Simple case:
- program $P$
- partition of the program variables of $P$ in
  - low security variables $low$ and
  - high security variables $high$

Definition (Non-interference – Version 1)

For program $P$ the high variables $high$ do not interfere with the low variables $low$

$\iff$

when starting $P$ with arbitrary values for $low$, then the values of $low$ after executing $P$, are independent of the choices of $high$. 
Non-Interference

Prominent information flow property: non-interference

Simple case:

- program \( P \)
- partition of the program variables of \( P \) in
  - low security variables \( low \) and
  - high security variables \( high \)

Definition (Non-interference – Version 1)

For program \( P \) the high variables \( high \) do not interfere with the low variables \( low \)

\[ \Leftrightarrow \]

when starting \( P \) with arbitrary values for \( low \), then the values of \( low \) after executing \( P \), are independent of the choices of \( high \).
Non-Interference

Prominent information flow property: non-interference

Simple case:

- program $P$
- partition of the program variables of $P$ in
  - low security variables $low$
  - high security variables $high$

Definition (Non-interference – Version 1)

For program $P$ the high variables $high$ do not interfere with the low variables $low$

\[ \Leftrightarrow \]

when starting $P$ with arbitrary values for $low$, then the values of $low$ after executing $P$, are independent of the choices of $high$. 
Non-Interference

Prominent information flow property: non-interference

Simple case:

- program $P$
- partition of the program variables of $P$ in
  - low security variables $low$
  - high security variables $high$

Definition (Non-interference – Version 1)

For program $P$ the high variables $high$ do not interfere with the low variables $low$

$\iff$

when starting $P$ with arbitrary values for $low$, then the values of $low$ after executing $P$, are independent of the choices of $high$. 
Non-Interference

Prominent information flow property: non-interference

Simple case:
- program $P$
- partition of the program variables of $P$ in
  - low security variables $\text{low}$ and
  - high security variables $\text{high}$

Definition (Non-interference – Version 2)

For program $P$ the high variables $\text{high}$ do not interfere with the low variables $\text{low}$

$\iff$

running two instances of $P$, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.
Non-Interference

Prominent information flow property: non-interference

Simple case:

- program $P$
- partition of the program variables of $P$ in
  - low security variables $low$
  - high security variables $high$

Definition (Non-interference – Version 2)

For program $P$ the high variables $high$ do not interfere with the low variables $low$

\[ \iff \]

running two instances of $P$, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.
Non-Interference

Prominent information flow property: non-interference

Simple case:

- program $P$
- partition of the program variables of $P$ in
  - low security variables $low$ and
  - high security variables $high$

Definition (Non-interference – Version 2)

For program $P$ the high variables $high$ do not interfere with the low variables $low$

$\iff$

running two instances of $P$, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.
Non-Interference

Prominent information flow property: non-interference

Simple case:
- program $P$
- partition of the program variables of $P$ in
  - low security variables $low$ and
  - high security variables $high$

Definition (Non-interference – Version 2)
For program $P$ the high variables $high$ do not interfere with the low variables $low$

$\iff$

running two instances of $P$, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.
Non-Interference

Prominent information flow property: non-interference

Simple case:

- program $P$
- partition of the program variables of $P$ in
  - low security variables $low$
  - high security variables $high$

Definition (Non-interference – Version 2)

For program $P$ the high variables $high$ do not interfere with the low variables $low$

$\iff$ running two instances of $P$, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.
Non-Interference

Prominent information flow property: non-interference

Simple case:
- program $P$
- partition of the program variables of $P$ in
  - low security variables $low$
  - high security variables $high$

Definition (Non-interference – Version 2)

For program $P$ the high variables $high$ do not interfere with the low variables $low$

$\iff$
unning two instances of $P$, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.
Non-Interference

Prominent information flow property: non-interference

Simple case:
- program $P$
- partition of the program variables of $P$ into
  - low security variables $low$
  - high security variables $high$

Definition (Non-interference – Version 2)

For program $P$ the high variables $high$ do not interfere with the low variables $low$

$\iff$

running two instances of $P$, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.
Examples

Which methods are safe?

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 safe?

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 save?

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 save?

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 safe?

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 safe?

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

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

Which methods are save?

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

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

Which methods are save?

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

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

Introduction  Formalising Non-Interference in JavaDL  Extending JML for Non-Interference Specifications  Generalising the Non-Interference Formalisation  Full Example  Summary

Definition (Non-interference)

For program $P$ the high variables $high$ do not interfere with the low variables $low$ if running two instances of $P$, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

$$\forall l_{in} \forall h_{in}^{1} \forall h_{in}^{2} \forall l_{out}^{1} \forall l_{out}^{2} (\{low := l_{in} || high := h_{in}^{1}\}[P]low = l_{out}^{1}$$
$$\land \{low := l_{in} || high := h_{in}^{2}\}[P]low = l_{out}^{2}$$
$$\rightarrow l_{out}^{1} = l_{out}^{2}$$
Formalising Non-Interference in JavaDL

Definition (Non-interference)

For program $P$ the high variables $high$ do not interfere with the low variables $low$

$\iff$

running two instances of $P$, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

$$
\forall l_{in} \forall h_{in} \forall h_{in} \forall l_{out} \forall l_{out} ( \\
\quad \{low := l_{in} \parallel high := h_{in}\}[P]low = l_{out}^1 \\
\quad \land \{low := l_{in} \parallel high := h_{in}\}[P]low = l_{out}^2 \\
\quad \rightarrow l_{out}^1 = l_{out}^2 )
$$
Definition (Non-interference)

For program $P$ the high variables $high$ do not interfere with the low variables $low$

$\iff$

running two instances of $P$, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

\[
\forall l_{in}\forall h_{in}\forall h_{in}^{2}\forall l_{out}^{1}\forall l_{out}^{2}(\quad \{ low := l_{in} || high := h_{in}^{1}\}[P]low = l_{out}^{1} \\
\wedge \{ low := l_{in} || high := h_{in}^{2}\}[P]low = l_{out}^{2} \\
\rightarrow l_{out}^{1} = l_{out}^{2} \quad )
\]
Definition (Non-interference)

For program \( P \) the high variables \( high \) do not interfere with the low variables \( low \)

\[ \iff \]

running two instances of \( P \), with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

\[
\forall l_{in} \forall h_{in} \forall h_{in}^2 \forall l_{out}^1 \forall l_{out}^2 \left( \begin{array}{l}
\{ low := l_{in} \ || \ high := h_{in}^1 \} [P] low = l_{out}^1 \\
\land \{ low := l_{in} \ || \ high := h_{in}^2 \} [P] low = l_{out}^2 \\
\rightarrow l_{out}^1 = l_{out}^2 \end{array} \right) \]
Definition (Non-interference)

For program $P$ the high variables $high$ do not interfere with the low variables $low$ if:

$$\forall l_{in} \forall h_{in} \forall h_{in} \forall l_{out} \forall l_{out} ( \{ low := l_{in} \parallel high := h_{in} \} [P] low = l_{out}^1$$

$$\land \{ low := l_{in} \parallel high := h_{in} \} [P] low = l_{out}^2$$

$$\rightarrow l_{out}^1 = l_{out}^2 )$$

Running two instances of $P$, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.
Definition (Non-interference)

For program $P$ the high variables $high$ do not interfere with the low variables $low$

$\iff$

running two instances of $P$, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

$$\forall l_{in} \forall h_{in}^1 \forall h_{in}^2 \forall l_{out}^1 \forall l_{out}^2 (\{low := l_{in} || high := h_{in}^1\}[P]low = l_{out}^1$$

$$\wedge \{low := l_{in} || high := h_{in}^2\}[P]low = l_{out}^2$$

$$\rightarrow l_{out}^1 = l_{out}^2)$$
Definition (Non-interference)

For program $P$ the high variables \textit{high} do not interfere with the low variables \textit{low} if

\[
\forall l_{in} \forall h_{in} \forall h_{in}^2 \forall l_{out}^1 \forall l_{out}^2 \forall \{ \text{low} := l_{in} \ || \ \text{high} := h_{in}^1 \}[P] \text{low} = l_{out}^1 \land \{ \text{low} := l_{in} \ || \ \text{high} := h_{in}^2 \}[P] \text{low} = l_{out}^2 
\rightarrow l_{out}^1 = l_{out}^2
\]

running two instances of $P$, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.
class SecurePasswordFile {
    private int[] names, passwords;
    //@ invariant names.length == passwords.length;
    public boolean check(int user, int password) {
        //@ loop.invariant ...
        for (int i = 0; i < names.length; i++) {
            if (names[i] == user &&
                passwords[i] == password) {
                return true;
            }
        }
        return false;
    }
}
class SecurePasswordFile {

private int[] names, passwords;
//@ invariant names.length == passwords.length;

public boolean check(int user, int password) {
    //@ loop_invvariant ...
    for (int i = 0; i < names.length; i++) {
        if (names[i] == user &&
            passwords[i] == password) {
            return true;
        }
    }
    return false;
}
Simple Example

class SecurePasswordFile {
  private int[] names, passwords;
  //@ invariant names.length == passwords.length;
  public boolean check(int user, int password) {
    //@ loop.invariant ...
    for (int i = 0; i < names.length; i++) {
      if (names[i] == user &&
          passwords[i] == password) {
        return true;
      }
    }
    return false;
  }
}
Simple Example

// General assumptions + class invariants //
wellFormed(heap1) ∧ ... 

// Symbolic execution //

∧ { heap := heap1 } 

\[\{ r = pwf.check(user, password); \} \]\n
r = outR1 

∧ { heap := heap2 } 

\[\{ r = pwf.check(user, password); \} \]\n
r = outR2 

// Comparision of the low variables //

→ outR1 = outR2
Simple Example

class SecurePasswordFile {

  private int[] names, passwords;
  //@ invariant names.length == passwords.length;

  public boolean check(int user, int password) {
    //@ loop_inv...
    for (int i = 0; i < names.length; i++) {
      if (names[i] == user && passwords[i] == password) {
        return true;
      }
    }
    return false;
  }
}

Introduction
Formalising Non-Interference in JavaDL
Extending JML for Non-Interference Specifications
Generalising the Non-Interference Formalisation
Full Example
Summary

Christoph Scheben – Deductive Verification of Information Flow Properties

July 13, 2011
How to define low and high variables in JML?

- Definition of low and high with respect to some security level.

**Definition (Security level)**

A security level is a set of heap locations.

- All heap locations of a security level are low with respect to that level, all other high.
- Definition of security levels in JML via model fields of type “location set”.
Example

```java
/*@ model \locset pwdFileManager;

@ accessible pwdFileManager: footprint;
@ represents pwdFileManager =

@ names, names[*], passwords, passwords[*];
@*/
```

Informal semantics:

- Set of locations defined by the evaluation of the model field in the current heap.
- Might evaluate to different security levels in different heaps.
Example

```java
/*@ normal_behavior
@ ... @ respects anyUser;
@*/
boolean check(int user, int password) { ... }
```

Informal semantics:
- Set of security levels for which a method fulfills the non-interference property.
JML – Parameter Dependencies

Example

```java
/*@ normal_behavior */
@ ...
@ secure_for checkUser, checkUser : checkUser;
/*@*/
boolean check(int user, int password) {
    ...
}
```

Informal semantics:
- **Parameter pre-condition:** the value which is passed to the method depends at most on the specified locations.
- **Return value post-condition:** the return value depends at most on the specified locations.
Example

```java
/*@ normal_behavior
@ ... @
@ declassify ( \exists int i; 0 <= i && i < names.length;
@ names[i] == user
@ && passwords[i] == password
@ )
@ \from pwdFileManager
@ \to checkUser
@ \if true ;
@*/

boolean check(int user, int password) { ... }
```
Informal semantics:

- Information to be declassified in form of a term or formula.
- May depend at most on the locations specified in the “from“ part.
- May flow at most to the locations specified in then “to” part.
- Declassification only if the “if” part evaluates to true (in the pre-heap).

Semantic form of declassification:

- Declassification is part of the method contract.
Full Example – JML Specification

```java
class SecurePasswordFile {

    /*@ model \locset checkUser;
    @ accessible checkUser: footprint;
    @ represents checkUser \such that
    @ \subset(checkUser, footprint);
    @

    @ model \locset anyUser;
    @ accessible anyUser: footprint;
    @ represents anyUser \such that
    @ \subset(anyUser, footprint);
    @

    @ invariant names.length == passwords.length;
    @*/

    private int[] names, passwords;
}
```
Full Example – JML Specification

```java
/*@ normal_behavior
   @ modifies \nothing;
   @ secure_for checkUser, checkUser:checkUser;
   @ respects anyUser;
   @ declassify ( \exists int i;
     @ 0 <= i && i < names.length;
     @ names[i] == user
     && passwords[i] == password
     @ )
   @ to checkUser;
   */

public boolean check(int user, int password) {
    ...
}
```
Generalising the JavaDL Formalisation

Simple version for program variables partitioned into high and low variables:

\[ \forall l_{in} \forall h_{in}^{1} \forall h_{in}^{2} \forall l_{out}^{1} \forall l_{out}^{2} ( \{ \text{low} := l_{in} \ || \ \text{high} := h_{in}^{1} \} [P] \text{low} = l_{out}^{1} \]
\[ \land \{ \text{low} := l_{in} \ || \ \text{high} := h_{in}^{2} \} [P] \text{low} = l_{out}^{2} \]
\[ \rightarrow l_{out}^{1} = l_{out}^{2} \)

Generalised version for arbitrary (definable) similarity relations \( \sim_{in} \) and \( \sim_{out} \) defined over program variables (heaps) \( h^{1} \) and \( h^{2} \):

\[ \forall h_{in}^{1} \forall h_{in}^{2} \forall h_{out}^{1} \forall h_{out}^{2} ( \{ \text{heap} := h_{in}^{1} \} [P] \text{heap} = h_{out}^{1} \]
\[ \land \{ \text{heap} := h_{in}^{2} \} [P] \text{heap} = h_{out}^{2} \]
\[ \land h_{in}^{1} \sim_{in} h_{in}^{2} \]
\[ \rightarrow h_{out}^{1} \sim_{out} h_{out}^{2} \)

Introduction  Formalising Non-Interference in JavaDL  Extending JML for Non-Interference Specifications  Generalising the Non-Interference Formalisation  Full Example  Summary

Christoph Scheben – Deductive Verification of Information Flow Properties  July 13, 2011  19/25
Generalising the JavaDL Formalisation

Where $h_{in}^1 \sim_{in} h_{in}^2$ has the form:

- All elements of the respects clause are low variables,

$$\forall Object \: o : \forall Field \: f : (o, f) \in \{heap := h_{in}^1\} \text{respects} \rightarrow \{heap := h_{in}^1\} o.f = \{heap := h_{in}^2\} o.f$$

- all parameters with dependencies $\subseteq$ respects are low and

$$\land \land_{i \in \{1..n_{par}\}} (\{heap := h_{in}^1\} (secure\_for_i \subseteq \text{respects}) \rightarrow par_i^1 = par_i^2)$$

- all declassifications with to-part $\subseteq$ respects are known.

$$\land \land_{i \in \{1..n_{decl}\}} (\{heap := h_{in}^1\} (to_i \subseteq \text{respects}) \rightarrow (\{heap := h_{in}^1\} \text{decl}_i \leftrightarrow \{heap := h_{in}^2\} \text{decl}_i))$$
Generalising the JavaDL Formalisation

Where $h^1_{out} \sim_{out} h^2_{out}$ has the form:

- All elements of the respects clause are low variables,

$$\forall \text{Object } o : \forall \text{Field } f : (o, f) \in \{\text{heap} := h^1_{in}\}\text{respects}$$
$$\rightarrow \{\text{heap} := h^1_{out}\} o.f = \{\text{heap} := h^2_{out}\} o.f$$

- all parameters with dependencies $\subseteq$ respects are low and

$$\land \bigwedge_{i \in \{1 \ldots n_{par}\}} (\{\text{heap} := h^1_{in}\}(\text{secure}_{for_i} \subseteq \text{respects}) \rightarrow \text{par}_i^1 = \text{par}_i^2)$$

- if the return dependencies $\subseteq$ respects, then return is low.

$$\land (\{\text{heap} := h^1_{in}\}(\text{secure}_{for_{return}} \subseteq \text{respects}) \rightarrow \text{return}^1 = \text{return}^2)$$
Full Example – JavaDL Formalisation

// General Assumptions + Class Invariants
2 wellFormed(heapAtPre1) \land \ldots

// Symbolic Execution
4 \land \{heap:=heapAtPre1\} \{\ldots

// Input Relation
6 \land equalsAtLocs(heapAtPre1, heapAtPre2,
8 \{heap:=heapAtPre1\} self.anyUser \cap \{\}
10 \land (\{heap:=heapAtPre1\} self.passwordFileUser \subseteq \{heap:=heapAtPre1\} self.anyUser
12 \rightarrow user1 = user2)
14 \ldots
// Input-Relation — Declassification

\[ \begin{align*}
& \land ( \{ \text{heap} := \text{heapAtPre1} \} \text{self.passwordFileUser} \\
& \subseteq \{ \text{heap} := \text{heapAtPre1} \} \text{self.anyUser} \\
& \rightarrow ( \{ \text{heap} := \text{heapAtPre1} \} \\
& \quad \exists \texttt{int } i0; \quad \begin{align*}
& & ( 0 \leq i0 \land i0 < \text{self.names.length} \\
& & \land \text{inInt}(i0) \\
& & \land \text{self.names}[i0] = \text{user1} \\
& & \land \text{self.passwords}[i0] = \text{password1} ) \end{align*} \\
& \leftrightarrow \{ \text{heap} := \text{heapAtPre2} \} \\
& \quad \exists \texttt{int } i1; \quad \begin{align*}
& & ( 0 \leq i1 \land i1 < \text{self.names.length} \\
& & \land \text{inInt}(i1) \\
& & \land \text{self.names}[i1] = \text{user1} \\
& & \land \text{self.passwords}[i1] = \text{password1} )
\end{align*} ) \end{align*} \]
Full Example – JavaDL Formalisation

// Output–Relation
32 → equalsAtLocs(heapAtPost1, heapAtPost2, 
{heap:=heapAtPre1} self.anyUser ∩ { })

34 ∧ ( {heap:=heapAtPre1} self.passwordFileUser 
36 ⊆ {heap:=heapAtPre1} self.anyUser 
38 → result1 = result2 )

40 ∧ ( {heap:=heapAtPre1} self.passwordFileUser 
42 ⊆ {heap:=heapAtPre1} self.anyUser 
44 → user1 = user2 )
Not tackled

- Comparison of objects.
- How to use information flow contracts.
- Quantitative analysis of specifications.