Applications of Formal Verification

Deductive Verification of Information Flow Properties of Java Programs

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov | SS 2012
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 $\textit{low}$ and
  - high-security variables $\textit{high}$
- In the following, non-interference means $\textit{high}$ do not interfere with $\textit{low}$ in $P$ (=no information flows from $\textit{high}$ to $\textit{low}$)

**Definition (Non-interference – not quite formal)**

When starting $P$ with arbitrary values for $\textit{low}$, the values of $\textit{low}$ after executing $P$ are independent of the choices of $\textit{high}$.
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 $low$
  - 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 $low$
  - 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

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

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;
    }
}
Which methods are secure?

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

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

Which methods are secure?

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

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

Which methods are secure?

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

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

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

**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 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 $r$.

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

- **Problem**: not suitable for automatic verification ⇨ instantiation of existential quantifier difficult.
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 \[\neg\neg\]
  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 $r$.

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

- **Problem**: not suitable for automatic verification $\leadsto$ instantiation of existential quantifier difficult.
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 $\Rightarrow$ 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 ⇞ instantiation of existential quantifier difficult.
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 $\iff$ 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.

$$\forall \text{in}_1 \forall \text{in}_2 \forall \text{out}_1 \forall \text{out}_2 \{ \text{low} := \text{in}_1 \} (\{ \text{high} := \text{in}_1 \}[P] \text{out}_1 = \text{low} \land \{ \text{high} := \text{in}_2 \}[P] \text{out}_2 = \text{low} \rightarrow \text{out}_1 = \text{out}_2)$$
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.

$$\forall \text{in}_l \forall \text{in}_h^1 \forall \text{in}_h^2 \forall \text{out}_l^1 \forall \text{out}_l^2 \{ \text{low} := \text{in}_l \}( \{ \text{high} := \text{in}_h^1 \}[P]\text{out}_l^1 = \text{low} \land \{ \text{high} := \text{in}_h^2 \}[P]\text{out}_l^2 = \text{low} \rightarrow \text{out}_l^1 = \text{out}_l^2 )$$
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.

$$\forall in \exists \text{low} \exists \text{out} \quad \{ \text{low} := \text{in} \}( \{ \text{high} := \text{in}_h^1 \}[P]\text{out}_i^1 = \text{low} \land \{ \text{high} := \text{in}_h^2 \}[P]\text{out}_i^2 = \text{low} \Rightarrow \text{out}_i^1 = \text{out}_i^2 )$$
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.

\[
\forall \text{in}_l \forall \text{in}_h^1 \forall \text{in}_h^2 \forall \text{out}_l^1 \forall \text{out}_l^2 \{ \text{low} := \text{in}_l \} ( \\
\{ \text{high} := \text{in}_h^1 \}[P] \text{out}_l^1 = \text{low} \\
\land \{ \text{high} := \text{in}_h^2 \}[P] \text{out}_l^2 = \text{low} \\
\rightarrow \text{out}_l^1 = \text{out}_l^2 
) 
\]
Let \( T(\text{high}, \text{low}) \) 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.
Let $T(\text{high, low})$ 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.
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.

$$\forall in_l \forall in^1_h \forall in^2_h \forall out^1_l \forall out^2_l \{ \text{low := in}_l \} (\{ \text{high := in}^1_h \} T = \{ \text{high := in}^2_h \} T$$

$$\land \{ \text{high := in}^1_h \}[P]out^1_l = \text{low}$$

$$\land \{ \text{high := in}^2_h \}[P]out^2_l = \text{low}$$

$$\rightarrow out^1_l = out^2_l$$
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.

$$\forall \text{in}_1 \forall \text{in}_h \forall \text{in}_h^2 \forall \text{out}_1 \forall \text{out}_2 \begin{cases} \text{low} := \text{in}_1 \end{cases} \{ \\
\{ \text{high} := \text{in}_h^1 \} T = \{ \text{high} := \text{in}_h^2 \} T \\
\land \{ \text{high} := \text{in}_h^1 \}[P]\text{out}_1^1 = \text{low} \\
\land \{ \text{high} := \text{in}_h^2 \}[P]\text{out}_2^2 = \text{low} \\
\rightarrow \text{out}_1^1 = \text{out}_2^2 \\
\}$$
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 $r$.

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

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

\[
\forall in_l \forall in^1_h \forall in^2_h \forall out^1_l \forall out^2_l \{ \text{low} := in_l \} ( \\
\{ \text{high} := in^1_h \} \langle P \rangle _{\text{true}} \land \{ \text{high} := in^2_h \} \langle P \rangle _{\text{true}} \land \\
(\{ \text{high} := in^1_h \} \langle P \rangle _{\text{out}^1_l} = \text{low} \land \\
\{ \text{high} := in^2_h \} \langle P \rangle _{\text{out}^2_l} = \text{low} \rightarrow \\
out^1_l = out^2_l ) \\
\lor (\{ \text{high} := in^1_h \} [P]_{\text{false}} \land \{ \text{high} := in^2_h \} [P]_{\text{false}}) \\
) 
\]
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.

\[
\forall \text{in}_l \forall \text{in}_h^1 \forall \text{in}_h^2 \forall \text{out}_l^1 \forall \text{out}_l^2 \left\{ \text{low} := \text{in}_l \right\} ( \\
\left\{ \text{high} := \text{in}_h^1 \right\} \langle P \rangle \text{true} \land \left\{ \text{high} := \text{in}_h^2 \right\} \langle P \rangle \text{true} \land \\
\left( \left\{ \text{high} := \text{in}_h^1 \right\} \langle P \rangle \text{out}_l^1 = \text{low} \land \\
\left\{ \text{high} := \text{in}_h^2 \right\} \langle P \rangle \text{out}_l^2 = \text{low} \rightarrow \\
\text{out}_l^1 = \text{out}_l^2 \right) \\
\lor \left( \left\{ \text{high} := \text{in}_h^1 \right\} [P] \text{false} \land \left\{ \text{high} := \text{in}_h^2 \right\} [P] \text{false} \right)
\]
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.

\[
\forall \text{in}_1\forall \text{in}_h^1\forall \text{in}_h^2\forall \text{out}_1^1\forall \text{out}_1^2 \ \{ \text{low} := \text{in}_1 \} ( \{ \text{high} := \text{in}_h^1 \} \langle P \rangle \text{true} \land \{ \text{high} := \text{in}_h^2 \} \langle P \rangle \text{true} \land ( \{ \text{high} := \text{in}_h^1 \} \langle P \rangle \text{out}_1^1 = \text{low} \land \{ \text{high} := \text{in}_h^2 \} \langle P \rangle \text{out}_1^2 = \text{low} \rightarrow \text{out}_1^1 = \text{out}_1^2 ) ) \lor ( \{ \text{high} := \text{in}_h^1 \} [P] \text{false} \land \{ \text{high} := \text{in}_h^2 \} [P] \text{false} )
\]
Adding Termination-sensitivity

Another encoding of termination-sensitive non-interf.

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

$$\forall in_l \{ low := in_l \}( \\
\exists in_h \{ high := in_h \} \langle P \rangle \text{true} \rightarrow \\
\exists r \forall in_h \{ high := in_h \} \langle P \rangle low = r $$
Not Covered Here

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