Content-Dependent Security Policies in Avionics

Tomasz Maciążek       Hanne Riis Nielson       Flemming Nielson

Department of Applied Mathematics and Computer Science
Technical University of Denmark
Richard Petersens Plads, Building 324
2800 Kongens Lyngby, Denmark

January 19, 2016
Outline

Background and motivation

The CBIF verification tool

Decentralized Label Model (DLM)

Content-dependent policies

Program validation – high level description

The CBIF tool in action

Conclusion
Background and motivation
Separation kernels and secure gateways are used in MILS to ensure separation and controlled communication between components.
Separation kernels and secure gateways are used in MILS to ensure separation and controlled communication between components.

Parts of secure gateways can be validated using static program analysis.
Background and motivation

- **Separation kernels** and **secure gateways** are used in **MILS** to ensure separation and controlled communication between components.
- Parts of secure gateways can be validated using **static program analysis**.
- **DLM** proves to be insufficient/too cumbersome to use.

**Idea:** DLM labels should be content-dependent for the gateway scenario.
Background and motivation

- **Separation kernels and secure gateways** are used in MILS to ensure separation and controlled communication between components.
- Parts of secure gateways can be validated using **static program analysis**.
- **DLM** proves to be insufficient/too cumbersome to use.
- Idea: **DLM labels** should be content-dependent for the gateway scenario.
Background and motivation

- Separation kernels and secure gateways are used in MILS to ensure separation and controlled communication between components.
- Parts of secure gateways can be validated using static program analysis.
- DLM proves to be insufficient/too cumbersome to use.
- Idea: DLM labels should be content-dependent for the gateway scenario.
- Policies – regulation of label assignment.
Separation kernels and secure gateways are used in MILS to ensure separation and controlled communication between components.

Parts of secure gateways can be validated using static program analysis.

DLM proves to be insufficient/too cumbersome to use.

Idea: DLM labels should be content-dependent for the gateway scenario.

Policies – regulation of label assignment.

A demultiplexer use case scenario by Müller et al. in article Secure Information Flow Control in Safety-Critical Systems.
The CBIF verification tool
CBIF

- A static validation tool written in C#
The CBIF verification tool

CBIF

▶ A static validation tool written in C#
▶ Uses ANTLR for parsing the input programs, and Microsoft’s Z3 for internal comparisons of policies against the possible program states
CBIF

- A static validation tool written in C#
- Uses ANTLR for parsing the input programs, and Microsoft’s Z3 for internal comparisons of policies against the possible program states
- Supports programs written in a subset of C as the input
The CBIF verification tool

CBIF

- A static validation tool written in C#
- Uses ANTLR for parsing the input programs, and Microsoft’s Z3 for internal comparisons of policies against the possible program states
- Supports programs written in a subset of C as the input
- Allows annotating the input code with conditional DLM-like policies
The CBIF verification tool

CBIF

- A static validation tool written in C#
- Uses ANTLR for parsing the input programs, and Microsoft’s Z3 for internal comparisons of policies against the possible program states
- Supports programs written in a subset of C as the input
- Allows annotating the input code with conditional DLM-like policies
- Provides output helpful in tracking down information flow problems
Decentralized Label Model (DLM)
Background information

- A labelling system for ensuring information flow security (confidentiality and integrity alike).

DLM elements:

- Principals – entities that can perform actions in the system; may act as owners, readers and writers of data.
- Labels – consist of principals; form the security policies that attached to variables in DLM

\[
\begin{align*}
O_1 &\rightarrow R_1 ; \\
&\ldots \\
O_n &\rightarrow R_n ; \\
O_1 &\leftarrow W_1 ; \\
&\ldots \\
O_n &\leftarrow W_n
\end{align*}
\]

Partial ordering, example:

\[
\{A \rightarrow B, C\} \sqsubseteq \{A \rightarrow B; C \rightarrow B\}
\]
Background information

- A labelling system for ensuring information flow security (confidentiality and integrity alike).
- First published in 1997 by Myers and Liskov.
A labelling system for ensuring information flow security (confidentiality and integrity alike).

First published in 1997 by Myers and Liskov.

Controlled downgrading of security labels.
Background information

- A labelling system for ensuring information flow security (confidentiality and integrity alike).
- First published in 1997 by Myers and Liskov.
- Controlled *downgrading* of security labels
  - Declassification for confidentiality
  - Endorsement for integrity
Background information

- A labelling system for ensuring information flow security (confidentiality and integrity alike).
- First published in 1997 by Myers and Liskov.
- Controlled downgrading of security labels
  - Declassification for confidentiality
  - Endorsement for integrity
- DLM elements:
A labelling system for ensuring *information flow security* (confidentiality and integrity alike).

First published in 1997 by Myers and Liskov.

Controlled *downgrading* of security labels
- Declassification for confidentiality
- Endorsement for integrity

DLM elements:
- Principals – entities that can perform actions in the system; may act as owners, readers and writers of data.
A labelling system for ensuring **information flow security** (confidentiality and integrity alike).

First published in 1997 by Myers and Liskov.

Controlled **downgrading** of security labels

- Declassification for confidentiality
- Endorsement for integrity

**DLM elements:**

- **Principals** – entities that can perform actions in the system; may act as owners, readers and writers of data.
- **Labels** – consist of principals; form the **security policies** that attached to variables in DLM

\[
\begin{align*}
O_1 &\rightarrow R_1; \ldots; O_n \rightarrow R_n; \ O_1 &\leftarrow W_1; \ldots; O_n &\leftarrow W_n
\end{align*}
\]
Decentralized Label Model (DLM)

Background information

- A labelling system for ensuring information flow security (confidentiality and integrity alike).
- First published in 1997 by Myers and Liskov.
- Controlled downgrading of security labels
  - Declassification for confidentiality
  - Endorsement for integrity
- DLM elements:
  - Principals – entities that can perform actions in the system; may act as owners, readers and writers of data.
  - Labels – consist of principals; form the security policies that attached to variables in DLM

\[
\{O_1 \rightarrow R_1; \ldots; O_n \rightarrow R_n; O_1 \leftarrow W_1; \ldots; O_n \leftarrow W_n\}
\]
Background information

- A labelling system for ensuring information flow security (confidentiality and integrity alike).
- First published in 1997 by Myers and Liskov.
- Controlled **downgrading** of security labels
  - Declassification for confidentiality
  - Endorsement for integrity
- DLM elements:
  - Principals – entities that can perform actions in the system; may act as owners, readers and writers of data.
  - Labels – consist of principals; form the security policies that attached to variables in DLM

\[
\{ O_1 \rightarrow R_1; \ldots; O_n \rightarrow R_n; O_1 \leftarrow W_1; \ldots; O_n \leftarrow W_n \}\]
Decentralized Label Model (DLM)

Background information

- A labelling system for ensuring information flow security (confidentiality and integrity alike).
- First published in 1997 by Myers and Liskov.
- Controlled downgrading of security labels
  - Declassification for confidentiality
  - Endorsement for integrity
- DLM elements:
  - Principals – entities that can perform actions in the system; may act as owners, readers and writers of data.
  - Labels – consist of principals; form the security policies that attached to variables in DLM

\[
\{ O_1 \rightarrow R_1; \ldots; O_n \rightarrow R_n; O_1 \leftarrow W_1; \ldots; O_n \leftarrow W_n \}\]
Background information

- A labelling system for ensuring information flow security (confidentiality and integrity alike).
- First published in 1997 by Myers and Liskov.
- Controlled downgrading of security labels
  - Declassification for confidentiality
  - Endorsement for integrity
- DLM elements:
  - Principals – entities that can perform actions in the system; may act as owners, readers and writers of data.
  - Labels – consist of principals; form the security policies that attached to variables in DLM

\[
\{ O_1 \rightarrow R_1; \ldots; O_n \rightarrow R_n; O_1 \leftarrow W_1; \ldots; O_n \leftarrow W_n \}\]
Background information

- A labelling system for ensuring information flow security (confidentiality and integrity alike).
- First published in 1997 by Myers and Liskov.
- Controlled *downgrading* of security labels
  - Declassification for confidentiality
  - Endorsement for integrity
- DLM elements:
  - Principals – entities that can perform actions in the system; may act as owners, readers and writers of data.
  - Labels – consist of principals; form the security policies that attached to variables in DLM

\[
\{O_1 \to R_1; \ldots; O_n \to R_n; O_1 \leftarrow W_1; \ldots; O_n \leftarrow W_n\}
\]
A labelling system for ensuring information flow security (confidentiality and integrity alike).

First published in 1997 by Myers and Liskov.

Controlled downgrading of security labels

- Declassification for confidentiality
- Endorsement for integrity

DLM elements:

- Principals – entities that can perform actions in the system; may act as owners, readers and writers of data.
- Labels – consist of principals; form the security policies that attached to variables in DLM

\[ \{ O_1 \rightarrow R_1; \ldots; O_n \rightarrow R_n; O_1 \leftarrow W_1; \ldots; O_n \leftarrow W_n \} \]
Decentralized Label Model (DLM)

Background information

- A labelling system for ensuring information flow security (confidentiality and integrity alike).
- First published in 1997 by Myers and Liskov.
- Controlled downgrading of security labels
  - Declassification for confidentiality
  - Endorsement for integrity
- DLM elements:
  - Principals – entities that can perform actions in the system; may act as owners, readers and writers of data.
  - Labels – consist of principals; form the security policies that attached to variables in DLM
    \[
    \{O_1 \rightarrow R_1; \ldots; O_n \rightarrow R_n; O_1 \leftarrow W_1; \ldots; O_n \leftarrow W_n\}
    \]
- Partial ordering
A labelling system for ensuring information flow security (confidentiality and integrity alike).

First published in 1997 by Myers and Liskov.

Controlled downgrading of security labels

- Declassification for confidentiality
- Endorsement for integrity

DLM elements:

- Principals – entities that can perform actions in the system; may act as owners, readers and writers of data.
- Labels – consist of principals; form the security policies that attached to variables in DLM

\[ \{O_1 \rightarrow R_1; \ldots; O_n \rightarrow R_n; O_1 \leftarrow W_1; \ldots; O_n \leftarrow W_n\} \]

Partial ordering, example: \( \{A \rightarrow B, C\} \sqsubseteq \{A \rightarrow B; C \rightarrow B\} \)
Content-dependent policies
Policy specification

- An extension to the DLM labels

```c
struct s {
    int det;
    int *data;
} {
    (self.det == 1 => self.data={{Alice->Bob}});
    (self.det == 2 => self.data={{Alice->Chuck}});
}
```
An extension to the DLM labels
Equality conditions on slots (identifiers o places in program’s memory)
Policy specification

- An extension to the DLM labels
- Equality conditions on slots (identifiers o places in program’s memory)
- Example

```c
struct s {
    int {{Alice->Bob, Chuck}} det;
    int *data;
};

(self.det == 1 => self.data={{Alice->Bob}});
(self.det == 2 => self.data={{Alice->Chuck}});
```
Policy specification

- An extension to the DLM labels
- Equality conditions on slots (identifiers o places in program’s memory)
- Example

```c
struct s {
    int {{Alice->Bob, Chuck}} det;
    int *data;
};

(self.det == 1 => self.data={{Alice->Bob}}); (self.det == 2 => self.data={{Alice->Chuck}})
```
Policy specification

- An extension to the DLM labels
- Equality conditions on slots (identifiers o places in program’s memory)
- Example

```c
struct s {
  int {{Alice->Bob,Chuck}} det;
  int *data;
};

(self.det == 1 => self.data={{Alice->Bob}});
(self.det == 2 => self.data={{Alice->Chuck}})
```
Policy specification

- An extension to the DLM labels
- Equality conditions on slots (identifiers o places in program’s memory)
- Example

```c
struct s {
    int {{Alice->Bob, Chuck}} det;
    int *data;
} {
    (self.det == 1 => self.data={{Alice->Bob}});
    (self.det == 2 => self.data={{Alice->Chuck}})
};
```
Policy specification

- An **extension** to the DLM labels
- Equality **conditions** on slots (identifiers o places in program’s memory)
- Example

```c
struct s {
    int {{Alice->Bob, Chuck}} det;
    int *data;
};

(self.det == 1 => self.data= {{Alice->Bob}});
(self.det == 2 => self.data= {{Alice->Chuck}})

```
Policy specification

- An extension to the DLM labels
- Equality conditions on slots (identifiers o places in program’s memory)
- Example

```c
struct s {
    int {{Alice->Bob, Chuck}} det;
    int *data;
};

(self.det == 1 => self.data={{Alice->Bob}});
(self.det == 2 => self.data={{Alice->Chuck}})
```
Global policy

- Formed from the individual policies of variables
Global policy

- Formed from the individual policies of variables
- Used internally in the CBIF tool
Global policy

- Formed from the individual policies of variables
- Used internally in the CBIF tool
- Present in the **formal type system** for reasoning about information flow security
Global policy

- Formed from the individual policies of variables
- Used internally in the CBIF tool
- Present in the formal type system for reasoning about information flow security
- Syntax

\[
P ::= X : L \\
\quad| \phi \Rightarrow P \\
\quad| P_1; P_2 \\
\phi ::= x = n \\
\quad| \phi_1 \land \phi_2 \\
\quad| \phi_1 \lor \phi_2
\]
Global policy

- Formed from the individual policies of variables
- Used internally in the CBIF tool
- Present in the **formal type system** for reasoning about information flow security
- Syntax

\[
P ::= X : L \\
\quad \quad \mid \phi \Rightarrow P \\
\quad \quad \mid P_1; P_2 \\
\phi ::= x = n \\
\quad \quad \mid \phi_1 \land \phi_2 \\
\quad \quad \mid \phi_1 \lor \phi_2
\]
Global policy

- Formed from the individual policies of variables
- Used internally in the CBIF tool
- Present in the formal type system for reasoning about information flow security
- Syntax

\[
\begin{align*}
P & ::= X : L \\
& \mid \phi \Rightarrow P \\
& \mid P_1; P_2 \\
\phi & ::= x = n \\
& \mid \phi_1 \land \phi_2 \\
& \mid \phi_1 \lor \phi_2
\end{align*}
\]
Global policy

- Formed from the individual policies of variables
- Used internally in the CBIF tool
- Present in the formal type system for reasoning about information flow security
- Syntax

\[ P ::= X : L \]
\[ \mid \phi \Rightarrow P \]
\[ \mid P_1 ; P_2 \]
\[ \phi ::= x = n \]
\[ \mid \phi_1 \land \phi_2 \]
\[ \mid \phi_1 \lor \phi_2 \]
Global policy

- Formed from the individual policies of variables
- Used internally in the CBIF tool
- Present in the formal type system for reasoning about information flow security
- Syntax

\[
P ::= X : L \quad \mid \phi \Rightarrow P \quad \mid P_1; P_2 \quad \phi ::= x = n \quad \mid \phi_1 \land \phi_2 \quad \mid \phi_1 \lor \phi_2
\]
Global policy

- Formed from the individual policies of variables
- Used internally in the CBIF tool
- Present in the **formal type system** for reasoning about information flow security
- Syntax

\[
P ::= X : L \\
| \phi \Rightarrow P \\
| P_1; P_2 \\
\phi ::= x = n \\
| \phi_1 \land \phi_2 \\
| \phi_1 \lor \phi_2
\]
Program validation – high level description
Relation to DLM

Several aspects of validation are similar to DLM.
Relation to DLM

Several aspects of validation are similar to DLM. Validating an assignment of form:

\[ x_v = e; \]
Relation to DLM

Several aspects of validation are similar to DLM. Validating an assignment of form:

\[ xv = e; \]

- The policy of \( xv \) (denoted \( xv \)) must be at least as restrictive as the aggregation of policies of variables from expression \( e \) (denoted \( e \)).
Relation to DLM

Several aspects of validation are similar to DLM. Validating an assignment of form:

\[ xv = e; \]

- The policy of \( xv \) (denoted \( xv \)) must be at least as restrictive as the aggregation of policies of variables from expression \( e \) (denoted \( e \))

\[ e \sqsubseteq xv \]
Relation to DLM

Several aspects of validation are similar to DLM. Validating an assignment of form:

\[ x_v = e; \]

- The policy of \( x_v \) (denoted \( x_v \)) must be at least as restrictive as the aggregation of policies of variables from expression \( e \) (denoted \( e \))
  \[ e \sqsubseteq x_v \]

- \( x_v \) must be also at least as restrictive as the policy of the program counter (\( pc \))
Relation to DLM

Several aspects of validation are similar to DLM. Validating an assignment of form:

\[ xv = e ; \]

- The policy of \( xv \) (denoted \( xv \)) must be at least as restrictive as the aggregation of policies of variables from expression \( e \) (denoted \( e \))

\[ e \sqsubseteq xv \]

- \( xv \) must be also at least as restrictive as the policy of the program counter (\( pc \))

\[ pc \sqsubseteq xv \]
Relation to DLM

Several aspects of validation are similar to DLM. Validating an assignment of form:

\[ x_v = e; \]

- The policy of \( x_v \) (denoted \( x_v \)) must be at least as restrictive as the aggregation of policies of variables from expression \( e \) (denoted \( e \))

\[ e \sqsubseteq x_v \]

- \( x_v \) must be also at least as restrictive as the policy of the program counter (\( pc \))

\[ pc \sqsubseteq x_v \]

The \( pc \) policy arises from the variables present in the conditions of the enclosing while loops and if conditionals.
The actual policies applying or possible to apply at any moment are determined by the **policy conditions** and the possible program **states**.
Content-dependency

- The actual policies applying or possible to apply at any moment are determined by the policy conditions and the possible program states.
- The possible states are determined by constraint environments resulting from previous statements and conditions on the enclosing block statements (if/while).
Content-depencency

- The actual policies applying or possible to apply at any moment are determined by the policy conditions and the possible program states.
- The possible states are determined by constraint environments resulting from previous statements and conditions on the enclosing block statements (if/while).
  - $\phi_{pc}$ – the constraint environment holding before the assignment
  - $\psi_{pc}$ – the constraint environment holding after the assignment
The actual policies applying or possible to apply at any moment are determined by the policy conditions and the possible program states.

The possible states are determined by constraint environments resulting from previous statements and conditions on the enclosing block statements (if/while).

- $\phi_{pc}$ – the constraint environment holding before the assignment
- $\psi_{pc}$ – the constraint environment holding after the assignment

The final validation formula is:

$$e_{\phi_{pc}} \sqcup p c_{\phi_{pc}} \sqsubseteq x v \psi_{pc}$$
Formal validation

In the type system and in the CBIF tool, for validating the fourth line in the following code snippet:

```c
1 int {{A->B}} x = 2;
2 int {( self == 1 => {A->B}) (self == 2 => {A->B,C})} y;
3 y = x;
```

Two policies would be created from the global policy, and compared on `y`:
Formal validation

In the type system and in the CBIF tool, for validating the fourth line in the following code snippet:

```c
int {{A->B}} x = 2;
int {(self == 1 => {A->B})
    (self == 2 => {A->B,C})} y;
y = x;
```

Two policies would be created from the global policy, and compared on $y$:

\[
P_{left} = (\text{true} \Rightarrow x, y : \{A \rightarrow B\});
(y = 1 \Rightarrow \emptyset : \{A \rightarrow B\});
(y = 2 \Rightarrow \emptyset : \{A \rightarrow B, C\})\]
Formal validation

In the type system and in the CBIF tool, for validating the fourth line in the following code snippet:

```
1 int {{A->B}} x = 2;
2 int {(self == 1 => {A->B})
3   (self == 2 => {A->B,C})} y;
4 y = x;
```

Two policies would be created from the global policy, and compared on y:

\[
P_{left} = (true \Rightarrow x, y : \{A \rightarrow B\});
(y = 1 \Rightarrow \emptyset : \{A \rightarrow B\});
(y = 2 \Rightarrow \emptyset : \{A \rightarrow B, C\})
\]

\[
P_{right} = (true \Rightarrow x : \{A \rightarrow B\});
(2 = 1 \Rightarrow y : \{A \rightarrow B\});
(2 = 2 \Rightarrow y : \{A \rightarrow B, C\})
\]
Formal validation

In the type system and in the CBIF tool, for validating the fourth line in the following code snippet:

```plaintext
1 int {A→B} x = 2;
2 int { (self == 1 => {A→B})
3    (self == 2 => {A→B,C}) } y;
4 y = x;
```

Two policies would be created from the global policy, and compared on `y`:

\[ P_{left} = (true \Rightarrow x, y : \{A \rightarrow B\}); \]
\[ (y = 1 \Rightarrow \emptyset : \{A \rightarrow B\}); \]
\[ (y = 2 \Rightarrow \emptyset : \{A \rightarrow B, C\}) \]

\[ P_{right} = (true \Rightarrow x : \{A \rightarrow B\}); \]
\[ (2 = 1 \Rightarrow y : \{A \rightarrow B\}); \]
\[ (2 = 2 \Rightarrow y : \{A \rightarrow B, C\}) \]
Formal validation

In the type system and in the CBIF tool, for validating the fourth line in the following code snippet:

1. \(\text{int } \{\text{A}\to\text{B}\} \ x = 2;\)
2. \(\text{int } \{(\text{self} == 1 \Rightarrow \{\text{A}\to\text{B}\}) \text{)}\)
3. \(\text{ (self} == 2 \Rightarrow \{\text{A}\to\text{B},\text{C}\})\} \ y;\)
4. \(y = x;\)

Two policies would be created from the global policy, and compared on \(y\):

\[P_{left} = (\text{true} \Rightarrow x, y : \{A \to B\});\]
\[(y = 1 \Rightarrow \emptyset : \{A \to B\});\]
\[(y = 2 \Rightarrow \emptyset : \{A \to B, C\});\]
\[P_{right} = (\text{true} \Rightarrow x : \{A \to B\});\]
\[(2 = 1 \Rightarrow y : \{A \to B\});\]
\[(2 = 2 \Rightarrow y : \{A \to B, C\});\]
The CBIF tool in action
Example

```
struct s {
  int {{Alice->Bob,Chuck}} det;
  int *data;
}

(self.det == 1 => self.data={Alice->Bob});
(self.det == 2 => self.data={Alice->Chuck})
};

struct s input;

int out_chan{
  (self.index == 0 => self={Alice->Bob});
  (self.index == 1 => self={Alice->Chuck})
} [2];

int counter = 0;
while(counter < 2)[counter >= 0] {
  if(input.det == counter) {
    out_chan[counter] = input.(*data);
  }
  counter = counter + 1;
}
```
Example

```c
struct s {
    int {{Alice->Bob,Chuck}} det;
    int *data;
}

(self.det == 1 => self.data={{Alice->Bob}});  
(self.det == 2 => self.data={{Alice->Chuck}})
};

struct s input;

int out_chan{
    (self.index == 0 => self={{Alice->Bob}}); 
    (self.index == 1 => self={{Alice->Chuck}})
} [2];

int counter = 0;
while(counter < 2)[counter >= 0] {
    if(input.det == counter) {
        out_chan[counter] = input.(*data);
    }
    counter = counter + 1;
}
```
Example

```c
struct s {
    int {{Alice->Bob,Chuck}} det;
    int *data;
};

(int self.det == 1 => self.data={Alice->Bob});
(int self.det == 2 => self.data={Alice->Chuck});

struct s input;

int out_chan{
(int self.index == 0 => self={Alice->Bob});
(int self.index == 1 => self={Alice->Chuck})
} [2];

int counter = 0;
while(counter < 2) [counter >= 0] {
    if(input.det == counter) {
        out_chan[counter] = input.(*data);
    }
    counter = counter + 1;
}
```
The CBIF tool in action

Example

```plaintext
struct s {
  int {{Alice->Bob,Chuck}} det;
  int *data;
}

(self.det == 1 => self.data={Alice->Bob});
(self.det == 2 => self.data={Alice->Chuck})
}

struct s input;

int out_chan{
  (self.index == 0 => self={Alice->Bob});
  (self.index == 1 => self={Alice->Chuck})
} [2];

int counter = 0;
while(counter < 2) [counter >= 0] {
  if(input.det == counter) {
    out_chan[counter] = input.(*data);
  }
  counter = counter + 1;
}
```

Tomasz Maciążek (DTU, Denmark)
The CBIF tool in action

Example

```c
struct s {
    int {{Alice->Bob,Chuck}} det;
    int *data;
}

(self.det == 1 => self.data={Alice->Bob});
(self.det == 2 => self.data={Alice->Chuck});
};

struct s input;

int out_chan{
    (self.index == 0 => self={Alice->Bob});
    (self.index == 1 => self={Alice->Chuck})
} [2];

int counter = 0;

while(counter < 2)[counter >= 0] {
    if(input.det == counter) {
        out_chan[counter] = input.(*data);
    }
    counter = counter + 1;
}
```
Example

```c
struct s {
    int {{Alice->Bob, Chuck}} det;
    int *data;
};

(struct s self) {
    if (self.det == 1) self.data = {Alice->Bob};
    if (self.det == 2) self.data = {Alice->Chuck};
}

struct s input;

int out_chan{
    if (self.index == 0) self = {Alice->Bob};
    if (self.index == 1) self = {Alice->Chuck};
} [2];

int counter = 0;
while (counter < 2) {
    if (input.det == counter) {
        out_chan[counter] = input.(*data);
    }
    counter = counter + 1;
}
```
Example

```c
struct s {
  int {{Alice->Bob,Chuck}} det;
  int *data;
}

  (self.det == 1 => self.data={Alice->Bob});
  (self.det == 2 => self.data={Alice->Chuck})
};

struct s input;

int out_chan{
  (self.index == 0 => self={Alice->Bob});
  (self.index == 1 => self={Alice->Chuck})
} [2];

int counter = 0;
while(counter < 2) [counter >= 0] {
  if(input.det == counter) {
    out_chan[counter] = input.(*data);
  }
  counter = counter + 1;
}
```
The CBIF tool in action

Example

```c
struct s {
    int {{Alice->Bob, Chuck}} det;
    int *data;
} {
    (self.det == 1 => self.data={Alice->Bob});
    (self.det == 2 => self.data={Alice->Chuck})
};
struct s input;

int out_chan{
    (self.index == 0 => self={Alice->Bob});
    (self.index == 1 => self={Alice->Chuck})
} [2];
int counter = 0;
while(counter < 2) [counter >= 0] {
    if(input.det == counter) {
        out_chan[counter] = input.(*data);
    }
    counter = counter + 1;
}
```

Tomasz Maciążek (DTU, Denmark)
Example

```c
struct s {
  int int {{Alice->Bob,Chuck}} det;
  int *data;
}
{
  (self.det == 1 => self.data={Alice->Bob});
  (self.det == 2 => self.data={Alice->Chuck})
};

struct s input;

int out_chan{
  (self.index == 0 => self={Alice->Bob});
  (self.index == 1 => self={Alice->Chuck})
} [2];

int counter = 0;
while(counter < 2) [counter >= 0] {
  if(input.det == counter) {
    out_chan[counter] = input.(*data);
  }
  counter = counter + 1;
}
```
Validation failed. Offending statement (line 17):
out_chan[counter] = input.data;

Reason:
LHS policy is more restrictive than RHS policy

LHS policy:
((...) =>
input.det|out_chan={Alice->Bob,Chuck};
((input.det == 2) => input.data|out_chan={Alice->Chuck});
((input.det == 1) => input.data|out_chan={Alice->Bob});
((out_chan.index == 1) => ={Alice->Chuck});
((out_chan.index == 0) => ={Alice->Bob});

RHS policy:
input.det={Alice->Bob,Chuck};
((input.det == 2) => input.data={Alice->Chuck});
((input.det == 1) => input.data={Alice->Bob});
((counter == 1) => out_chan={Alice->Chuck});
((counter == 0) => out_chan={Alice->Bob})

Model:
out_chan:{Alice} ->[input.det=1, out_chan.index=1, counter=1]
Validation failed. Offending statement (line 17):
out_chan[counter] = input.data;

Reason:
LHS policy is more restrictive than RHS policy

LHS policy:
((... ) =>
inpu t.det | out_chan={Alice ->Bob, Chuck};
((input.det == 2) => input.data | out_chan={Alice ->Chuck});
((input.det == 1) => input.data | out_chan={Alice ->Bob});
((out_chan.index == 1) => ={Alice ->Chuck});
((out_chan.index == 0) => ={Alice ->Bob}))

RHS policy:
inpu t.det={Alice ->Bob, Chuck};
((input.det == 2) => input.data={Alice ->Chuck});
((input.det == 1) => input.data={Alice ->Bob});
((counter == 1) => out_chan={Alice ->Chuck});
((counter == 0) => out_chan={Alice ->Bob})

Model:
out_chan:{Alice} ->[input.det=1, out_chan.index=1, counter=1]
Validation failed. Offending statement (line 17):
out_chan[counter] = input.data;

Reason:
LHS policy is more restrictive than RHS policy

LHS policy:
((...) =>
input.det|out_chan={Alice ->Bob, Chuck};
((input.det == 2) => input.data|out_chan={Alice ->Chuck});
((input.det == 1) => input.data|out_chan={Alice ->Bob});
((out_chan.index == 1) => ={Alice ->Chuck});
((out_chan.index == 0) => ={Alice ->Bob}))

RHS policy:
input.det={Alice ->Bob, Chuck};
((input.det == 2) => input.data={Alice ->Chuck});
((input.det == 1) => input.data={Alice ->Bob});
((counter == 1) => out_chan={Alice ->Chuck});
((counter == 0) => out_chan={Alice ->Bob})

Model:
out_chan:{Alice} ->[input.det=1, out_chan.index=1, counter=1]
Validation failed. Offending statement (line 17):
out_chan[counter] = input.data;

Reason:
LHS policy is more restrictive than RHS policy

LHS policy:
((...) =>
input.det|out_chan={Alice->Bob,Chuck};
((input.det == 2) => input.data|out_chan={Alice->Chuck});
((input.det == 1) => input.data|out_chan={Alice->Bob});
((out_chan.index == 1) => ={Alice->Chuck});
((out_chan.index == 0) => ={Alice->Bob}))

RHS policy:
input.det={Alice->Bob,Chuck};
((input.det == 2) => input.data={Alice->Chuck});
((input.det == 1) => input.data={Alice->Bob});
((counter == 1) => out_chan={Alice->Chuck});
((counter == 0) => out_chan={Alice->Bob})

Model:
out_chan:{Alice} ->[input.det=1, out_chan.index=1, counter=1]
Validation failed. Offending statement (line 17):
out_chan[counter] = input.data;

Reason:
LHS policy is more restrictive than RHS policy

LHS policy:
((...)) =>
input.det|out_chan={Alice->Bob, Chuck};
((input.det == 2) => input.data|out_chan={Alice->Chuck});
((input.det == 1) => input.data|out_chan={Alice->Bob});
((out_chan.index == 1) => ={Alice->Chuck});
((out_chan.index == 0) => ={Alice->Bob})

RHS policy:
input.det={Alice->Bob, Chuck};
((input.det == 2) => input.data={Alice->Chuck});
((input.det == 1) => input.data={Alice->Bob});
((counter == 1) => out_chan={Alice->Chuck});
((counter == 0) => out_chan={Alice->Bob})

Model:
out_chan:{Alice} ->[input.det=1, out_chan.index=1, counter=1]
CBIF output

1 Validation failed. Offending statement (line 17):
2 out_chan[counter] = input.data;
3
4 Reason:
5 LHS policy is more restrictive than RHS policy
6
7 LHS policy:
8 ( (...) =>
9 input.det | out_chan = { Alice -> Bob , Chuck };
10 (( input.det == 2 ) => input.data | out_chan = { Alice -> Chuck });
11 (( input.det == 1 ) => input.data | out_chan = { Alice -> Bob });
12 (( out_chan.index == 1 ) => = { Alice -> Chuck });
13 (( out_chan.index == 0 ) => = { Alice -> Bob }))
14
15 RHS policy:
16 input.det = { Alice -> Bob , Chuck };
17 (( input.det == 2 ) => input.data = { Alice -> Chuck });
18 (( input.det == 1 ) => input.data = { Alice -> Bob });
19 (( counter == 1 ) => out_chan = { Alice -> Chuck });
20 (( counter == 0 ) => out_chan = { Alice -> Bob })
21
22 Model:
23 out_chan: { Alice } -> [ input.det = 1 , out_chan.index = 1 , counter = 1 ]
Validation failed. Offending statement (line 17):
```
out_chan[counter] = input.data;
```

Reason:
LHS policy is more restrictive than RHS policy

LHS policy:
```
((... ) =>
input.det| out_chan={Alice->Bob,Chuck};
((input.det == 2) => input.data|out_chan={Alice->Chuck});
((input.det == 1) => input.data| out_chan={Alice->Bob});
((out_chan.index == 1) => ={Alice->Chuck});
((out_chan.index == 0) => ={Alice->Bob}))
```

RHS policy:
```
input.det={Alice->Bob,Chuck};
((input.det == 2) => input.data={Alice->Chuck});
((input.det == 1) => input.data={Alice->Bob});
((counter == 1) => out_chan={Alice->Chuck});
((counter == 0) => out_chan={Alice->Bob})
```

Model:
```
out_chan:{Alice} ->[input.det=1, out_chan.index=1, counter=1]
```
Validation failed. Offending statement (line 17):
```
out_chan[counter] = input.data;
```

Reason:
LHS policy is more restrictive than RHS policy

LHS policy:
```
((...) =>
input.det|out_chan={Alice->Bob,Chuck};
((input.det == 2) => input.data|out_chan={Alice->Chuck});
((input.det == 1) => input.data|out_chan={Alice->Bob}));
((out_chan.index == 1) => ={Alice->Chuck});
((out_chan.index == 0) => ={Alice->Bob}))
```

RHS policy:
```
input.det={Alice->Bob,Chuck};
((input.det == 2) => input.data={Alice->Chuck});
((input.det == 1) => input.data={Alice->Bob});
((counter == 1) => out_chan={Alice->Chuck});
((counter == 0) => out_chan={Alice->Bob})
```

Model:
```
out_chan:{Alice} ->[input.det=1, out_chan.index=1, counter=1]
```
Validation failed. Offending statement (line 17):
```c
out_chan[counter] = input.data;
```

Reason:
LHS policy is more restrictive than RHS policy

LHS policy:
```c
((...) =>
  input.det|out_chan={Alice->Bob,Chuck};
  ((input.det == 2) => input.data|out_chan={Alice->Chuck});
  ((input.det == 1) => input.data|out_chan={Alice->Bob});
  ((out_chan.index == 1) => ={Alice->Chuck});
  ((out_chan.index == 0) => ={Alice->Bob}))
```

RHS policy:
```c
input.det={Alice->Bob,Chuck};
((input.det == 2) => input.data={Alice->Chuck});
((input.det == 1) => input.data={Alice->Bob});
((counter == 1) => out_chan={Alice->Chuck});
((counter == 0) => out_chan={Alice->Bob})
```

Model:
```c
out_chan:{Alice} ->[input.det=1, out_chan.index=1, counter=1]
```
Example revised

Problem with `out_chan` for `input.det=1` and `counter=1`.

```c
struct s {
  int {{Alice->Bob, Chuck}} det;
  int *data;
};

struct s input;

int out_chan{
  (self.index == 0 => self={Alice->Bob});
  (self.index == 1 => self={Alice->Chuck})
} [2];

int counter = 0;
while(counter < 2) [counter >= 0] {
  if(input.det == counter) {
    out_chan[counter] = input.(*data);
  }
  counter = counter + 1;
}
```
Example revised

Problem with **out-chan** for **input.det=1** and **counter=1**.

```c
struct s { 
  int {{Alice -> Bob, Chuck}} det; 
  int *data; 
}; 

(struct s input; 

int out_chan{ 
  (self.index == 0 => self = {Alice -> Bob}); 
  (self.index == 1 => self = {Alice -> Chuck}) 
} [2]; 

int counter = 0; 
while (counter < 2) [counter >= 0] { 
  if (input.det == counter) { 
    out_chan[counter] = input.(*data); 
  } 
  counter = counter + 1; 
}
```
Example revised

Problem with `out_chan` for `input.det=1` and `counter=1`.

```c
struct s {
  int {{Alice->Bob, Chuck}} det;
  int *data;
}

(struct s input;

int out_chan{
  (self.index == 0 => self={Alice->Bob});
  (self.index == 1 => self={Alice->Chuck})
} [2];

int counter = 0;
while(counter < 2)[counter >= 0] {
  if(input.det == counter) {
    out_chan[counter] = input.(*data);
  }
  counter = counter + 1;
}
```
Example revised

Problem with `out_chan` for `input.det=1` and `counter=1`.

```c
struct s {
  int {{Alice->Bob,Chuck}} det;
  int *data;
}

  if(self.det == 1 => self.data={Alice->Bob});
  (self.det == 2 => self.data={Alice->Chuck})

};

struct s input;

int outChan{
  (self.index == 0 => self={Alice->Bob});
  (self.index == 1 => self={Alice->Chuck})
} [2];

int counter = 0;

while(counter < 2) [counter >= 0] { 
  if(input.det == counter) {
    out_chan[counter] = input.(*data);
  }
  counter = counter + 1;
}```
Example revised

Problem with `out_chan` for `input.det=1` and `counter=1`.

```c
struct s {
    int {{Alice->Bob,Chuck}} det;
    int *data;
}s{
    (self.det == 1 => self.data={Alice->Bob});
    (self.det == 2 => self.data={Alice->Chuck})
};
struct s input;

int out_chan{
    (self.index == 0 => self={Alice->Bob});
    (self.index == 1 => self={Alice->Chuck})
};
int counter = 0;
while(counter < 2)[counter >= 0] {
    if(input.det == counter) {
        out_chan[counter] = input.(*data);
    }
    counter = counter + 1;
}
```
Example revised

Problem with `out_chan` for `input.det=1` and `counter=1`.

```c
struct s {
  int {{Alice->Bob,Chuck}} det;
  int *data;
} {
  (self.det == 1 => self.data={Alice->Bob});
  (self.det == 2 => self.data={Alice->Chuck})
};
struct s input;

int out_chan{
  (self.index == 0 => self={Alice->Bob});
  (self.index == 1 => self={Alice->Chuck})
} [2];
int counter = 0;
while(counter < 2)[counter >= 0] {
  if(input.det == counter) {
    out_chan[counter] = input.(*data);
  }
  counter = counter + 1;
}
```

Tomasz Maciążek (DTU, Denmark)
Example revised

Problem with `out_chan` for `input.det=1` and `counter=1`.

```c
1 struct s {
2   int {{Alice->Bob,Chuck}} det;
3   int *data;
4 }{
5   (self.det == 1 => self.data={Alice->Bob});
6   (self.det == 2 => self.data={Alice->Chuck})
7 };
8 struct s input;
9
10 int out_chan{
11   (self.index == 0 => self={Alice->Bob});
12   (self.index == 1 => self={Alice->Chuck})
13 } [2];
14 int counter = 0;
15 while(counter < 2) [counter >= 0] {
16   if(input.det == counter + 1) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```
Conclusion
CBIF is a proof of concept that content-based analysis can be done on C-like programs.
Conclusion

- CBIF is a proof of concept that content-based analysis can be done on C-like programs.
- The tool has great performance but the complexity grows rapidly with size of the program, policies and number of principals.
CBIF is a proof of concept that content-based analysis can be done on C-like programs.

The tool has great performance but the complexity grows rapidly with size of the program, policies and number of principals.

Tools like this may reduce costs of certification of critical systems.
Conclusion

- CBIF is a proof of concept that content-based analysis can be done on C-like programs.
- The tool has great performance but the complexity grows rapidly with size of the program, policies and number of principals.
- Tools like this may reduce costs of certification of critical systems.
- There’s still a lot of work to be done.
CBIF is a proof of concept that content-based analysis can be done on C-like programs.

The tool has great performance but the complexity grows rapidly with size of the program, policies and number of principals.

Tools like this may reduce costs of certification of critical systems.

There’s still a lot of work to be done
  
  Extension of the supported subset of C
CBIF is a proof of concept that content-based analysis can be done on C-like programs.

The tool has great performance but the complexity grows rapidly with size of the program, policies and number of principals.

Tools like this may reduce costs of certification of critical systems.

There’s still a lot of work to be done:
- Extension of the supported subset of C
- Optimization
CBIF is a proof of concept that content-based analysis can be done on C-like programs.

The tool has great performance but the complexity grows rapidly with size of the program, policies and number of principals.

Tools like this may reduce costs of certification of critical systems.

There’s still a lot of work to be done:
- Extension of the supported subset of C
- Optimization
- Introduction of polymorphism
Conclusion

- CBIF is a proof of concept that content-based analysis can be done on C-like programs.
- The tool has great performance but the complexity grows rapidly with size of the program, policies and number of principals.
- Tools like this may reduce costs of certification of critical systems.
- There’s still a lot of work to be done:
  - Extension of the supported subset of C
  - Optimization
  - Introduction of polymorphism
  - Integration with external policy specification systems
Recap

Background and motivation

The CBIF verification tool

Decentralized Label Model (DLM)

Content-dependent policies

Program validation – high level description

The CBIF tool in action

Conclusion
Thank you for your attention
Semantics of DLM labels

- Partial ordering:

\[ L_1 \sqsubseteq L_2 \text{ iff } \forall p : \text{readers}(L_1, p) \supseteq \text{readers}(L_2, p) \]
\[ \land \text{writers}(L_1, p) \subseteq \text{writers}(L_2, p) \]

\[
\text{readers}(O \to R, p) = \begin{cases} 
\{p\} \cup R & \text{if } p \in O \\
\text{PRIN} & \text{otherwise}
\end{cases}
\]

\[
\text{readers}(L_1; L_2, p) = \text{readers}(L_1, p) \cap \text{readers}(L_2, p)
\]

\[
\text{writers}(O \leftarrow W, p) = \begin{cases} 
\{p\} \cup W & \text{if } p \in O \\
\emptyset & \text{otherwise}
\end{cases}
\]

\[
\text{writers}(L_1; L_2, p) = \text{writers}(L_1, p) \cup \text{writers}(L_2, p)
\]
Semantics of DLM labels

- Partial ordering:

  \[ L_1 \trianglelefteq L_2 \text{ iff } \forall p : \text{readers}(L_1, p) \supseteq \text{readers}(L_2, p) \]
  \[ \land \text{writers}(L_1, p) \subseteq \text{writers}(L_2, p) \]

- Example

  \[ \{ A \rightarrow B, C \} \trianglelefteq \{ A \rightarrow B; C \rightarrow B \} \]