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

Background and motivation

- Separation kernels and secure gateways are used in MILS to ensure separation and controlled communication between components

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

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

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

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
- 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#

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

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

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).

Background information

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

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

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:

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.

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

$$\{O_1 \rightarrow R_1; ...; O_n \rightarrow R_n; O_1 \leftarrow W_1; ...; O_n \leftarrow W_n\}$$

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

$$\{O_1 \rightarrow R_1; ...; O_n \rightarrow R_n; O_1 \leftarrow W_1; ...; O_n \leftarrow W_n\}$$

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

$$\{O_1 \rightarrow R_1; ...; O_n \rightarrow R_n; O_1 \leftarrow W_1; ...; O_n \leftarrow W_n\}$$

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

$$\{O_1 \rightarrow R_1; ...; O_n \rightarrow R_n; O_1 \leftarrow W_1; ...; O_n \leftarrow W_n\}$$

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

$$\{O_1 \rightarrow R_1; ...; O_n \rightarrow R_n; O_1 \leftarrow W_1; ...; O_n \leftarrow W_n\}$$

- Partial ordering

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

$$\{O_1 \rightarrow R_1; ...; O_n \rightarrow R_n; O_1 \leftarrow W_1; ...; O_n \leftarrow W_n\}$$

- 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

Policy specification

- 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

```
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 };
```

Policy specification

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

```
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 };
```

Policy specification

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

```
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 };
```

Policy specification

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

```
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 };
```

Policy specification

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

```
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 };
```

Policy specification

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

```
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 };
```

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

$$
\begin{aligned}
\mathtt{P} &::= X : L \\
&\mid \phi \Rightarrow \mathtt{P} \\
&\mid \mathtt{P}_1 ; \mathtt{P}_2 \\
\phi &::= x = n \\
&\mid \phi_1 \wedge \phi_2 \\
&\mid \phi_1 \vee \phi_2
\end{aligned}
$$

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

$$
\begin{aligned}
\mathtt{P} &::= X : L \\
&\mid \phi \Rightarrow \mathtt{P} \\
&\mid \mathtt{P}_1 ; \mathtt{P}_2 \\
\phi &::= x = n \\
&\mid \phi_1 \wedge \phi_2 \\
&\mid \phi_1 \vee \phi_2
\end{aligned}
$$

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

$$
\begin{aligned}
\mathtt{P} &::= X : L \\
&\mid \phi \Rightarrow \mathtt{P} \\
&\mid \mathtt{P}_1 ; \mathtt{P}_2 \\
\phi &::= x = n \\
&\mid \phi_1 \wedge \phi_2 \\
&\mid \phi_1 \vee \phi_2
\end{aligned}
$$

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

$$
\begin{aligned}
\mathtt{P} &::= X : L \\
&\mid \phi \Rightarrow \mathtt{P} \\
&\mid \mathtt{P}_1 ; \mathtt{P}_2 \\
\phi &::= x = n \\
&\mid \phi_1 \wedge \phi_2 \\
&\mid \phi_1 \vee \phi_2
\end{aligned}
$$

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

$$
\begin{aligned}
\mathtt{P} &::= X : L \\
&\mid \phi \Rightarrow \mathtt{P} \\
&\mid \mathtt{P}_1 ; \mathtt{P}_2 \\
\phi &::= x = n \\
&\mid \phi_1 \wedge \phi_2 \\
&\mid \phi_1 \vee \phi_2
\end{aligned}
$$

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

$$
\begin{aligned}
\mathtt{P} &::= X : L \\
&\mid \phi \Rightarrow \mathtt{P} \\
&\mid \mathtt{P}_1 ; \mathtt{P}_2 \\
\phi &::= x = n \\
&\mid \phi_1 \wedge \phi_2 \\
&\mid \phi_1 \vee \phi_2
\end{aligned}
$$

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:

```
xv = e;
```

Relation to DLM

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

```
xv = e;
```

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

Relation to DLM

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

```
xv = e;
```

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

$$\underline{e} \sqsubseteq \underline{xv}$$

Relation to DLM

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

```
xv = e;
```

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

$$\underline{e} \sqsubseteq \underline{xv}$$

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

Relation to DLM

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

```
xv = e;
```

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

$$\underline{e} \sqsubseteq \underline{xv}$$

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

$$\underline{pc} \sqsubseteq \underline{xv}$$

Relation to DLM

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

```
xv = e;
```

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

$$\underline{e} \sqsubseteq \underline{xv}$$

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

$$\underline{pc} \sqsubseteq \underline{xv}$$

The $\underline{pc}$ policy arises from the variables present in the conditions of the enclosing *while* loops and *if* conditionals

Content-depencency

- The actual policies applying or possible to apply at any moment are determined by the policy conditions and the possible program states.

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*).

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*).
  - ϕ_{pc} – the constraint environment holding *before* the assignment
  - ψ_{pc} – the constraint environment holding *after* the assignment

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*).
  - ϕ_{pc} – the constraint environment holding *before* the assignment
  - ψ_{pc} – the constraint environment holding *after* the assignment
- The final validation formula is:

$$\underline{e}_{\phi_{pc}} \sqcup \underline{pc}_{\phi_{pc}} \sqsubseteq \underline{xv}_{\psi_{pc}}$$

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:

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:

$$\begin{aligned} \mathrm{P}_{\mathit{left}} = (\mathbf{true} \Rightarrow x, y : \{A \to B\}); \\ (y = 1 \Rightarrow \emptyset : \{A \to B\}); \\ (y = 2 \Rightarrow \emptyset : \{A \to B, C\}) \end{aligned}$$

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:

$$\begin{aligned}
\mathbb{P}_{left} = &(\mathbf{true} \Rightarrow x, y : \{A \to B\});\\
&(y = 1 \Rightarrow \emptyset : \{A \to B\});\\
&(y = 2 \Rightarrow \emptyset : \{A \to B, C\})\\
\mathbb{P}_{right} = &(\mathbf{true} \Rightarrow x : \{A \to B\});\\
&(2 = 1 \Rightarrow y : \{A \to B\});\\
&(2 = 2 \Rightarrow y : \{A \to B, C\})
\end{aligned}$$

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:

$$\begin{aligned}
\mathbb{P}_{left} = &(\mathbf{true} \Rightarrow x, y : \{A \to B\});\\
&(y = 1 \Rightarrow \emptyset : \{A \to B\});\\
&(y = 2 \Rightarrow \emptyset : \{A \to B, C\})\\
\mathbb{P}_{right} = &(\mathbf{true} \Rightarrow x : \{A \to B\});\\
&(2 = 1 \Rightarrow y : \{A \to B\});\\
&(2 = 2 \Rightarrow y : \{A \to B, C\})
\end{aligned}$$

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:

$$\begin{aligned}
\mathbb{P}_{left} = &(\mathbf{true} \Rightarrow x, y : \{A \to B\});\\
&(y = 1 \Rightarrow \emptyset : \{A \to B\});\\
&(y = 2 \Rightarrow \emptyset : \{A \to B, C\})\\
\mathbb{P}_{right} = &(\mathbf{true} \Rightarrow x : \{A \to B\});\\
&(2 = 1 \Rightarrow y : \{A \to B\});\\
&(2 = 2 \Rightarrow y : \{A \to B, C\})
\end{aligned}$$

The CBIF tool in action

Example

```
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) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

Example

```
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) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

Example

```
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) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

Example

```
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) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

Example

```
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) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

Example

```
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) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

Example

```
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) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

Example

```
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) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

Example

```
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) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

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]
```

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]
```

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]
```

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]
```

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]
```

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]
```

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]
```

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]
```

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]
```

Example revised

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

```
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) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

Example revised

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

```
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) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

Example revised

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

```
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) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

Example revised

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

```
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) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

Example revised

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

```
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) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

Example revised

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

```
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) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

Example revised

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

```
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

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.

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

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

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

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

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

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:

$$L_1 \sqsubseteq L_2 \text{ iff } \forall p : \text{readers}(L_1, p) \supseteq \text{readers}(L_2, p) \\ \wedge \text{writers}(L_1, p) \subseteq \text{writers}(L_2, p)$$

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

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

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

$$\text{writers}(L_1; L_2, p) = \text{writers}(L_1, p) \cup \text{writers}(L_2, p)$$

Semantics of DLM labels

- Partial ordering:

$$L_1 \sqsubseteq L_2 \text{ iff } \forall p : \text{readers}(L_1, p) \supseteq \text{readers}(L_2, p) \\ \wedge \text{writers}(L_1, p) \subseteq \text{writers}(L_2, p)$$

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

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

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

$$\text{writers}(L_1; L_2, p) = \text{writers}(L_1, p) \cup \text{writers}(L_2, p)$$

- Example

$$\{A \to B, C\} \sqsubseteq \{A \to B; C \to B\}$$