Analysing Cryptographically-Masked Information Flows in MILS-AADL Specifications

Thomas Noll, Louis Wachtmeister
MOVES Söllerhaus Workshop
January 22–27, 2017
Context: The D-MILS Project

Content

Context: The D-MILS Project

Information Flow Security

The Slicing Approach

Handling Cryptographic Operations
D-MiLS Design Flow

Architectural Refinement

MILS AADL → MILS AADL → MILS AADL

Security Analysis | Safety Analysis | Performance Analysis

D-MiLS Design Flow

App A
Level B
Classified

App B
Level C
Unclassified

App C
Level A
Top Secret

Configurations / Schedules / Communication Routes

Node1 → Node2 → Node3

MILS Technical Platform

Configuration Compiler

Impementes/Satisfies

Autofocus Model → Autocode

Simulink Model → Autocode

Ada Code
The architecture expresses an interaction policy among a collection of components.

Circles represent architectural components (subjects / objects)

Arrows represent interactions

The absence of an arrow is as significant as the presence of one

Trusted Subject

Components are assumed to perform the functions specified by the architect (trusted components enforce a local policy)

Suitability of the architecture for some purpose presumes that the architect’s assumptions are met in the implementation of the architecture diagram.
Information Flow Security

Content

Context: The D-MILS Project

Information Flow Security

The Slicing Approach

Handling Cryptographic Operations
Information Flow Security

Information Flows

Two components

Component 1:
- Low IN
- High IN
- Low OUT
- High OUT

Component 2:
- Low IN
- High IN
- Low OUT
- High OUT
Information Flow Security

Information Flows

Legal Flows
Information Flow Security

Information Flows

Illegal Flows

Component 1

Low IN

High IN

Low OUT

High OUT

Component 2

Low IN

High IN

Low OUT

High OUT

Analyzing Cryptographically-Masked Information Flows in MILS-AADL Specifications

Thomas Noll

MOVE Söllershaus Workshop; January 22–27, 2017
Information Flow Security

Information Flows

Cryptographically-Masked Flows

Component 1
Low IN
High IN
Low OUT
High OUT

Component 2
Low IN
High IN
Low OUT
High OUT

6 of 27
Analysing Cryptographically-Masked Information Flows in MILS-AADL Specifications
Thomas Noll
MOVES Söllerhaus Workshop; January 22–27, 2017
Information Flow Security

Non-Interference

- Here: two security levels \( L \) (low/public) and \( H \) (high/confidential/secret/private)
  - partial order \( L \sqsubseteq H \) ("can flow to")
  - extension to multi-level security by generalisation to lattice
Information Flow Security

Non-Interference

- Here: two security levels $L$ (low/public) and $H$ (high/confidential/secret/private)
  - partial order $L \sqsubseteq H$ (“can flow to”)
  - extension to multi-level security by generalisation to lattice
- Analysis (can be) based on event traces in $Evt^*$
  - security assignment $\sigma: Evt \rightarrow \{L, H\}$
  - projection $t|_E$ for $t \in Evt^*$, $E \subseteq Evt$
  - $t_1, t_2 \in Evt^*$ called $E$-equivalent ($t_1 \sim_E t_2$) iff $t_1|_E = t_2|_E$

Definition (Non-interference [Goguen/Meseguer 1982])

Let $Evt = In \cup Out$ and $T \subseteq Evt^*$. Security assignment $\sigma$ ensures (event) non-interference if, for all $t_1, t_2 \in T$,

$$t_1 \sim_{In \cap \sigma^{-1}(L)} t_2 \Rightarrow t_1 \sim_{Out \cap \sigma^{-1}(L)} t_2$$

Interpretation: behaviour seen by “low” observer unaffected by changes in “high” behaviour (similar definition based on accesses to data elements)
Information Flow Security

Non-Interference

- Here: two security levels $L$ (low/public) and $H$ (high/confidential/secret/private)
  - partial order $L \sqsubseteq H$ (“can flow to”)
  - extension to multi-level security by generalisation to lattice
- Analysis (can be) based on event traces in $Evt^*$
  - security assignment $\sigma : Evt \rightarrow \{L, H\}$
  - projection $t|_E$ for $t \in Evt^*$, $E \subseteq Evt$
  - $t_1, t_2 \in Evt^*$ called $E$-equivalent $(t_1 \sim_E t_2)$ iff $t_1|_E = t_2|_E$

Definition (Non-interference [Goguen/Meseguer 1982])

Let $Evt = In \cup Out$ and $T \subseteq Evt^*$. Security assignment $\sigma$ ensures (event) non-interference if, for all $t_1, t_2 \in T$,

$$t_1 \sim_{In \cap \sigma^{-1}(L)} t_2 \implies t_1 \sim_{Out \cap \sigma^{-1}(L)} t_2$$

Interpretation: behaviour seen by “low” observer unaffected by changes in “high” behaviour (similar definition based on accesses to data elements)
Information Flow Security

Cryptographically-Masked Information Flow

- **Observation:** encryption breaks traditional non-interference
- Public ciphertexts *do* depend on confidential contents!

Example (Password encryption)

- \( \text{In} = \{ \text{pwd1}, \text{pwd2} \} \)
- \( \text{Out} = \{ \text{enc1}, \text{enc2} \} \)
- \( t_1 = \text{pwd1} \cdot \text{enc1} \)
- \( t_2 = \text{pwd2} \cdot \text{enc2} \)

Yields

- \( t_1 | \text{In} \cap \sigma^{-1}(L) = \epsilon = t_2 | \text{In} \cap \sigma^{-1}(L) \)
- But \( t_1 | \text{Out} \cap \sigma^{-1}(L) = \text{enc1} \neq \text{enc2} = t_2 | \text{Out} \cap \sigma^{-1}(L) \)

⇒ Interference

Goal

Find relaxed notion of (non-)interference that distinguishes between

- breaking non-interference because of legitimate use of (sufficiently strong) encryption and
- breaking non-interference due to an (unintended) leak

8 of 27 Analysing Cryptographically-Masked Information Flows in MILS-AADL Specifications
Thomas Noll
MOVES Söllerhaus Workshop; January 22–27, 2017
Information Flow Security

Cryptographically-Masked Information Flow

- **Observation:** encryption breaks traditional non-interference
- Public ciphertexts *do* depend on confidential contents!

<table>
<thead>
<tr>
<th>Example (Password encryption)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In = {pwd1_H, pwd2_H}, Out = {enc1_L, enc2_L}</td>
</tr>
<tr>
<td>t_1 = pwd1 \cdot enc1, t_2 = pwd2 \cdot enc2</td>
</tr>
<tr>
<td>Yields t_1</td>
</tr>
<tr>
<td>\Rightarrow Interference</td>
</tr>
</tbody>
</table>

Goal

Find relaxed notion of (non-)interference that distinguishes between
- breaking non-interference because of legitimate use of (sufficiently strong) encryption and
- breaking non-interference due to an (unintended) leak
Information Flow Security

Cryptographically-Masked Information Flow

- **Observation:** encryption breaks traditional non-interference
- Public ciphertexts *do* depend on confidential contents!

Example (Password encryption)

- \(\text{In} = \{\text{pwd}_1^H, \text{pwd}_2^H\}\), \(\text{Out} = \{\text{enc}_1^L, \text{enc}_2^L\}\)
- \(t_1 = \text{pwd}_1 \cdot \text{enc}_1, t_2 = \text{pwd}_2 \cdot \text{enc}_2\)
- Yields \(t_1|_{\text{In} \cap \sigma^{-1}(\text{L})} = \varepsilon = t_2|_{\text{In} \cap \sigma^{-1}(\text{L})}\), but \(t_1|_{\text{Out} \cap \sigma^{-1}(\text{L})} = \text{enc}_1 \neq \text{enc}_2 = t_2|_{\text{Out} \cap \sigma^{-1}(\text{L})}\)
  \(\Rightarrow\) Interference

Goal

Find relaxed notion of (non-)interference that distinguishes between
- breaking non-interference because of legitimate use of (sufficiently strong) encryption and
- breaking non-interference due to an (unintended) leak
Adapting Non-Interference

• Non-interference: if a program is run in two low-equivalent environments, the resulting environments are low-equivalent
• Confidentiality thus requires: attacker cannot distinguish between ciphertexts
• Naive approach: all ciphertexts are indistinguishable
• But: enables occlusion (i.e., security leaks by implicit data flow)
Information Flow Security

Adapting Non-Interference

- Non-interference: if a program is run in two low-equivalent environments, the resulting environments are low-equivalent
- Confidentiality thus requires: attacker cannot distinguish between ciphertexts
- Naive approach: all ciphertexts are indistinguishable
- But: enables occlusion (i.e., security leaks by implicit data flow)

Example (Occlusion)

\[
\begin{align*}
\text{m0} & \rightarrow \text{[then low1 := encrypt(val1, key1)]} \rightarrow \text{m1}; \\
\text{m1} & \rightarrow \text{[when high then low2 := encrypt(val2, key2)]} \rightarrow \text{m2}; \\
\text{m1} & \rightarrow \text{[when not high then low2 := low1]} \rightarrow \text{m2};
\end{align*}
\]

Cannot distinguish between low1 and low2 even though (in-)equality reflects high
Information Flow Security

Adapting Non-Interference

- Non-interference: if a program is run in two low-equivalent environments, the resulting environments are low-equivalent
- Confidentiality thus requires: attacker cannot distinguish between ciphertexts
- Naive approach: all ciphertexts are indistinguishable
- But: enables occlusion (i.e., security leaks by implicit data flow)

Example (Occlusion)

m0 -[then low1 := encrypt(val1, key1)]-> m1;
m1 -[when high then low2 := encrypt(val2, key2)]-> m2;
m1 -[when not high then low2 := low1]-> m2;

Cannot distinguish between low1 and low2 even though (in-)equality reflects high

Wanted: notion of low-equivalence that semantically rejects occlusion without preventing intuitively secure uses
Information Flow Security

Possibilistic Non-Interference [McCullough 1988]

- Encryption non-deterministically calculates a ciphertext out of a set
- Encrypted values low-equivalent (i.e., indistinguishable to attacker) if sets of possible results coincide
Information Flow Security

Possibilistic Non-Interference [McCullough 1988]

- Encryption non-deterministically calculates a ciphertext out of a set
- Encrypted values low-equivalent (i.e., indistinguishable to attacker) if sets of possible results coincide

Definition (Low-equivalence relation)

\[ \sim_L \] is a low-equivalence relation on ciphertexts if for all values \( v_1, v_2 \in Val \) and keys \( k_1, k_2 \in Val \):

1. safe usage: \( \forall u_1 \in encrypt(v_1, k_1). \exists u_2 \in encrypt(v_2, k_2) : u_1 \sim_L u_2 \)
2. exclude occlusion: \( \exists u_1 \in encrypt(v_1, k_1), u_2 \in encrypt(v_2, k_2) : u_1 \not\sim_L u_2 \)
Information Flow Security

Possibilistic Non-Interference [McCullough 1988]

- Encryption non-deterministically calculates a ciphertext out of a set
- Encrypted values low-equivalent (i.e., indistinguishable to attacker) if sets of possible results coincide

**Definition (Low-equivalence relation)**

\(\sim_L\) is a low-equivalence relation on ciphertexts if for all values \(v_1, v_2 \in Val\) and keys \(k_1, k_2 \in Val\):

1. safe usage: \(\forall u_1 \in \text{encrypt}(v_1, k_1) \exists u_2 \in \text{encrypt}(v_2, k_2) : u_1 \sim_L u_2\)
2. exclude occlusion: \(\exists u_1 \in \text{encrypt}(v_1, k_1), u_2 \in \text{encrypt}(v_2, k_2) : u_1 \not\sim_L u_2\)

Lifted to low-equivalence relation \(\sim_L\) on

- non-encrypted values \(v_1, v_2 \in Val\): \(v_1 \sim_L v_2\) iff \(v_1 = v_2\)
- environments \(\eta_1, \eta_2 : Dat \rightarrow Val\): \(\eta_1 \sim_L \eta_2\) iff \(\eta_1(d) \sim_L \eta_2(d)\) for all \(d \in Dat\) with \(\sigma(d) = L\)
Information Flow Security

Possibilistic Non-Interference II

Definition (Possibilistic non-interference)

Given a program $c$, security assignment $\sigma : Dat \rightarrow \{L, H\}$ ensures possibilistic non-interference if

\[
\forall \eta_1, \eta_2, \eta'_1 : Dat \rightarrow Val \text{ with } \eta_1 \sim_L \eta_2 \text{ and } \eta_1 \xrightarrow{c} \eta'_1
\]

\[
\exists \eta'_2 : Dat \rightarrow Val \text{ such that } \eta_2 \xrightarrow{c} \eta'_2 \text{ and } \eta'_1 \sim_L \eta'_2.
\]

Interpretation: If a program is run in two low-equivalent environments, there exists a possibility that each environment produced from the first environment is low-equivalent to some that can be produced from the second environment.
Possibilistic Non-Interference and Safe Usage of Encryption

Example (Safe usage of encryption)

\[ \text{m0} \rightarrow \text{m1}; \]

- Let \( \sigma(\text{high}) = H \) and \( \sigma(\text{key}) = \sigma(\text{low}) = L \)
- Let environments \( \eta_1, \eta_2 \) with \( \eta_1 \sim_L \eta_2 \) such that
  1. \( \eta_1(\text{high}) = v_1, \eta_1(\text{key}) = k \)
  2. \( \eta_2(\text{high}) = v_2, \eta_2(\text{key}) = k \)
- Execution respectively yields environment sets
  1. \( E'_1 = \{ \eta_1[\text{low} \mapsto u] \mid u \in \text{encrypt}(v_1, k) \} \)
  2. \( E'_2 = \{ \eta_2[\text{low} \mapsto u] \mid u \in \text{encrypt}(v_2, k) \} \)

- Now
  \[ (1) \forall u_1 \in \text{encrypt}(v_1, k). \exists u_2 \in \text{encrypt}(v_2, k) : u_1 \sim_L u_2 \]

implies that \( \forall \eta'_1 \in E'_1. \exists \eta'_2 \in E'_2 : \eta'_1 \sim_L \eta'_2 \)

\( \Rightarrow \) Possibilistic non-interference
Information Flow Security

Possibilistic Non-Interference and Occlusion

Example (Occlusion)

\[ m_0 - \text{[then low1 := encrypt(val, key)]} \rightarrow m_1; \]
\[ m_1 - \text{[when high then low2 := encrypt(val, key)]} \rightarrow m_2; \]
\[ m_1 - \text{[when not high then low2 := low1]} \rightarrow m_2; \]

- Let \( \sigma(\text{high}) = \sigma(\text{val}) = H \) and \( \sigma(\text{key}) = \sigma(\text{low1}) = \sigma(\text{low2}) = L \)
- Let environments \( \eta_1, \eta_2 \) with \( \eta_1 \sim_L \eta_2 \) such that
  1. \( \eta_1(\text{high}) = \text{true}, \eta_1(\text{val}) = v_1, \eta_1(\text{key}) = k \)
  2. \( \eta_2(\text{high}) = \text{false}, \eta_2(\text{val}) = v_2, \eta_2(\text{key}) = k \)
- Execution respectively yields
  1. \( E'_1 = \{ \eta_1[\text{low1} \mapsto u_1, \text{low2} \mapsto u_2] | u_1 \in \text{encrypt}(v_1, k), u_2 \in \text{encrypt}(v_2, k) \} \)
  2. \( E'_2 = \{ \eta_2[\text{low1} \mapsto u, \text{low2} \mapsto u] | u \in \text{encrypt}(v_1, k) \} \)
- Now (2) \( \exists u_1 \in \text{encrypt}(v_1, k), u_2 \in \text{encrypt}(v_2, k): u_1 \not\sim_L u_2 \)
  implies that \( \exists \eta'_1 \in E'_1 : \eta'_1(\text{low1}) \not\sim_L \eta'_1(\text{low2}) \)
- On the other hand, \( \forall \eta'_2 \in E'_2 : \eta'_2(\text{low1}) \sim_L \eta'_2(\text{low2}) \)
- Thus \( \exists \eta'_1 \in E'_1 . \forall \eta'_2 \in E'_2 : \eta'_1 \not\sim_L \eta'_2 \)
  \( \Rightarrow \) Possibilistic interference
Information Flow Security

Analysing Possibilistic Non-Interference

**Wanted**

Analysis algorithm for possibilistic non-interference that is

- **language-based**
  - no physical side channels, just logical system description
- **static**
  - based on textual specification, not on (dynamic) execution
- **sound**
  - all security leaks must be detected
- **precise**
  - as few false alarms as possible
The Slicing Approach

Content

Context: The D-MILS Project

Information Flow Security

The Slicing Approach

Handling Cryptographic Operations
The Slicing Approach

Slicing

Non-interference: which high inputs influence which low outputs?
Slicing: which outputs depend on which inputs?

- interesting output values define slicing criterion
- backward analysis of information flow based on program dependence graph

Applications:
- Debugging
- Testing
- Model checking
- Software security [Snelting et al.]

– relation to (classical) non-interference: if no high variable in the backward slice of any low output, then system is non-interfering
– interprocedural extension by context-sensitive slicing
The Slicing Approach

Slicing

Non-interference: which high inputs influence which low outputs?
Slicing: which outputs depend on which inputs?
  • interesting output values define slicing criterion
  • backward analysis of information flow based on program dependence graph

Applications:
  • Debugging
  • Testing
  • Model checking
  • Software security [Snelting et al.]
    – relation to (classical) non-interference: if no high variable in the backward slice of any low output, then system is non-interfering
    – interprocedural extension by context-sensitive slicing
The Slicing Approach

Slicing AADL Specifications for Model Checking [Odenbrett/Nguyen/Noll 2010]

\[ D := S; \ E := \emptyset; \ M := \emptyset; \ ] \quad \text{Initialization based on slicing criterion } S \subseteq Dat

\text{repeat}

\text{for all } m, g, f 
\rightarrow m' \in Trn \text{ with } 
\exists d \in D: \ f \text{ updates } d \text{ or } 
\exists d \in D: \ d \text{ inactive in } m \text{ but active in } m' \text{ or } 
\text{for all } e \in E \text{ do}

\quad M := M \cup \{ m \};

\text{Transitions that affect interesting data elements or have interesting triggers}

\text{for all } a \rightarrow d' \in Flw \text{ with } d' \in D \text{ do}

\quad D := D \cup \{ d \in Dat \mid g \text{ reads } d \} \cup \{ d \in Dat \mid f \text{ updates some } d' \in D \text{ reading } d \};

\quad M := M \cup \{ m \in Mod \mid a \rightarrow d' \text{ active in } m \};

\text{Transitions from/to interesting modes}

\text{for all } e \rightarrow e' \in Con \text{ with } e \in E \text{ or } e' \in E \text{ do}

\quad E := E \cup \{ e, \ e' \};

\quad M := M \cup \{ m \in Mod \mid e \rightarrow e' \text{ active in } m \};

\text{Connections involving interesting event ports}

\text{until nothing changes;}

\text{return } (D, E, M)
The Slicing Approach

Slicing AADL Specifications for Model Checking [Odenbrett/Nguyen/Noll 2010]

\[
D := S; \quad E := \emptyset; \quad M := \emptyset; \]
Initialization based on slicing criterion \( S \subseteq \text{Dat} \)

repeat
  for all \( m \xrightarrow{e.g. f} m' \in \text{Trn} \) with \( \exists d \in D : f \) updates \( d \)
    or \( \exists d \in D : d \) inactive in \( m \) but active in \( m' \)
    or \( e \in E \) do
    \( M := M \cup \{m\} \);

Transitions that affect interesting data elements or have interesting triggers

until nothing changes;

return \((D, E, M)\)
The Slicing Approach

Slicing AADL Specifications for Model Checking [Odenbrett/Nguyen/Noll 2010]

\[ D := S; E := \emptyset; M := \emptyset; \] \{ Initialization based on slicing criterion \( S \subseteq \text{Dat} \)

repeat
  for all \( m \xrightarrow{e.g.l} m' \in \text{Trn} \) with \( \exists d \in D : f \text{ updates } d \)
    or \( \exists d \in D : d \text{ inactive in } m \text{ but active in } m' \)
    or \( e \in E \) do
    \( M := M \cup \{ m \}; \)

  for all \( m \xrightarrow{e.g.l} m' \in \text{Trn} \) with \( m \in M \) or \( m' \in M \) do
    \( D := D \cup \{ d \in \text{Dat} \mid g \text{ reads } d \} \)
    \( \cup \{ d \in \text{Dat} \mid f \text{ updates some } d' \in D \text{ reading } d \}; \)
    \( E := E \cup \{ e \}; \)
    \( M := M \cup \{ m \}; \)
  \}

Transitions that affect interesting data elements or have interesting triggers

Transitions from/to interesting modes

until nothing changes;
return \((D, E, M)\)
The Slicing Approach

Slicing AADL Specifications for Model Checking [Odenbrett/Nguyen/Noll 2010]

\[ D := S; \quad E := \emptyset; \quad M := \emptyset; \]  
\{ Initialization based on slicing criterion \( S \subseteq Dat \)}

repeat
  for all \( m \xrightarrow{e.g.f} m' \in Trn \) with \( \exists d \in D : f \) updates \( d \)
  or \( \exists d \in D : d \) inactive in \( m \) but active in \( m' \)
  or \( e \in E \) do
    \( M := M \cup \{m\}; \)
  end for

  for all \( m \xrightarrow{e.g.f} m' \in Trn \) with \( m \in M \) or \( m' \in M \) do
    \( D := D \cup \{d \in Dat \mid g \) reads \( d\}\)
    \( \cup \{d \in Dat \mid f \) updates some \( d' \in D \) reading \( d\}\};\)
    \( E := E \cup \{e\}; \)
    \( M := M \cup \{m\}; \)
  end for

  for all \( a \leadsto d' \in Flw \) with \( d' \in D \) do
    \( D := D \cup \{d \in Dat \mid a \) reads \( d\}\};\)
    \( M := M \cup \{m \in Mod \mid a \leadsto d' \) active in \( m\};\)
  end for

until nothing changes;

return \((D, E, M)\)
The Slicing Approach

Slicing AADL Specifications for Model Checking [Odenbrett/Nguyen/Noll 2010]

\[ D := S; E := \emptyset; M := \emptyset; \] \hspace{1em} \text{Initialization based on slicing criterion } S \subseteq Dat

\textbf{repeat}
\hspace{1em} \textbf{for all } m \xrightarrow[e.g.]{} m' \in Trn \text{ with } \exists d \in D : f \text{ updates } d \\
\hspace{2em} \text{ or } \exists d \in D : d \text{ inactive in } m \text{ but active in } m' \\
\hspace{2em} \text{ or } e \in E \text{ do}
\hspace{2em} M := M \cup \{m\};

\hspace{1em} \textbf{for all } m \xrightarrow[e.g.]{} m' \in Trn \text{ with } m \in M \text{ or } m' \in M \text{ do}
\hspace{2em} D := D \cup \{d \in Dat \mid g \text{ reads } d\} \\
\hspace{2em} \quad \cup \{d \in Dat \mid f \text{ updates some } d' \in D \text{ reading } d\};
\hspace{2em} E := E \cup \{e\};
\hspace{2em} M := M \cup \{m\};

\hspace{1em} \textbf{for all } a \leadsto d' \in Flw \text{ with } d' \in D \text{ do}
\hspace{2em} D := D \cup \{d \in Dat \mid a \text{ reads } d'\};
\hspace{2em} M := M \cup \{m \in Mod \mid a \leadsto d' \text{ active in } m\};

\hspace{1em} \textbf{for all } e \leadsto e' \in Con \text{ with } e \in E \text{ or } e' \in E \text{ do}
\hspace{2em} E := E \cup \{e, e'\};
\hspace{2em} M := M \cup \{m \in Mod \mid e \leadsto e' \text{ active in } m\};

\textbf{until} nothing changes;
\textbf{return} \ (D, E, M)
The Slicing Approach

Slicing and Non-Interference

Theorem ([Wachtmeister 2016])

Let

- \( \sigma : \text{Dat} \cup \text{Evt} \cup \text{Mod} \rightarrow \{L, H\} \) a security assignment,
- \( S \subseteq \text{Dat} \) with \( \sigma(S) = \{L\} \), and
- \((D, E, M)\) the backward slice of \( S \).

If

\[ \sigma(D) \cup \sigma(E) \cup \sigma(M) = \{L\}, \]

then \( \sigma \) ensures (Goguen/Meseguer) non-interference.
The Slicing Approach

Example: A Crypto Controller
The Slicing Approach

Slicing the Crypto Controller

```plaintext
system cryptocontroller(
    inframe: in data (int,int)
    outframe: out data (int,enc int)
    mykeys: key pair
)

system split(
    frame: in data (int,int)
    header: out data int
    payload: out data int
    m0: initial mode
    m0 -[then header := frame[0];
        payload := frame[1]]-> m0
)

system bypass(
    inheader: in data int
    outheader: out data int
    m0: initial mode
    m0 -[then outheader := inheader]-> m0
)

system crypto(
    inpayload: in data int 0
    outpayload: out data enc int
    k: key pub(mykeys)
    m0: initial mode
    m0 -[then outpayload := encrypt(inpayload,k)]-> m0
)

system merge(
    header: in data int
    payload: in data enc int
    frame: out data (int,enc int)
    m0: initial mode
    m0 -[then frame := (header,payload)]-> m0
)

flow inframe -> split.frame
flow split.header -> bypass.inheader
flow split.payload -> crypto.inpayload
flow bypass.outheader -> merge.header
flow crypto.outpayload -> merge.payload
flow merge.frame -> outframe
```

Analysing Cryptographically-Masked Information Flows in MILS-AADL Specifications

Thomas Noll
MOVES Söllerhaus Workshop; January 22–27, 2017
The Slicing Approach

Slicing the Crypto Controller

system cryptocontroller(
    inframe: in data (int,int)
    outframe: out data (int,enc int)
    mykeys: key pair
)

system split(
    frame: in data (int,int)
    header: out data int
    payload: out data int
    m0: initial mode
    m0 -[then header := frame[0];
        payload := frame[1]]-> m0
)

system bypass(
    inheader: in data int
    outheader: out data int
    m0: initial mode
    m0 -[then outheader := inheader]-> m0
)

Slicing criterion: \{outframe\}

system crypto(
    inpayload: in data int 0
    outpayload: out data enc int
    k: key pub(mykeys)
    m0: initial mode
    m0 -[then outpayload := encrypt(inpayload,k)]-> m0
)

system merge(
    header: in data int
    payload: in data enc int
    frame: out data (int,enc int)
    m0: initial mode
    m0 -[then frame := (header,payload)]-> m0
)

flow inframe -> split.frame
flow split.header -> bypass.inheader
flow split.payload -> crypto.inpayload
flow bypass.outheader -> merge.header
flow crypto.outpayload -> merge.payload
flow merge.frame -> outframe

20 of 27 Analysing Cryptographically-Masked Information Flows in MILS-AADL Specifications
Thomas Noll
MOVES Söllerhaus Workshop; January 22–27, 2017
The Slicing Approach

Slicing the Crypto Controller

system cryptocontroller(
  inframe: in data (int,int)
outframe: out data (int,enc int)
mykeys: key pair
)

system split(
  frame: in data (int,int)
header: out data int
payload: out data int
m0: initial mode
m0 -> [then header := frame[0];
        payload := frame[1]]-> m0
)

system bypass(
  inheader: in data int
outheader: out data int
m0: initial mode
m0 -> [then outheader := inheader]-> m0
)

Add sources and modes of flows with interesting targets

system crypto(
  inpayload: in data int 0
outpayload: out data enc int
k: key pub(mykeys)
m0: initial mode
m0 -> [then outpayload := encrypt(inpayload,k)]-> m0
)

system merge(
  header: in data int
payload: in data enc int
frame: out data (int,enc int)
m0: initial mode
m0 -> [then frame := (header,payload)]-> m0
)

flow inframe -> split.frame
flow split.header -> bypass.inheader
flow split.payload -> crypto.inpayload
flow bypass.outheader -> merge.header
flow crypto.outpayload -> merge.payload
flow merge.frame -> outframe

20 of 27 Analysing Cryptographically-Masked Information Flows in MILS-AADL Specifications
Thomas Noll
MOVES Söllerhaus Workshop; January 22–27, 2017
The Slicing Approach

Slicing the Crypto Controller

```plaintext
system cryptocontroller(
    inframe: in data (int,int)
    outframe: out data (int,enc int)
    mykeys: key pair
)

system split(
    frame: in data (int,int)
    header: out data int
    payload: out data int
    m0: initial mode
    m0 -[then header := frame[0];
        payload := frame[1]]-> m0
)

system bypass(
    inheader: in data int
    outheader: out data int
    m0: initial mode
    m0 -[then outheader := inheader] -> m0
)

Add source modes of transitions that affect interesting data elements
```

```plaintext
system crypto(
    inpayload: in data int 0
    outpayload: out data enc int
    k: key pub(mykeys)
    m0: initial mode
    m0 -[then outpayload := encrypt(inpayload,k)]-> m0
)

system merge(
    header: in data int
    payload: in data enc int
    frame: out data (int,enc int)
    m0: initial mode
    m0 -[then frame := (header,payload)] -> m0
)

flow inframe -> split.frame
flow split.header -> bypass.inheader
flow split.payload -> crypto.inpayload
flow bypass.outheader -> merge.header
flow crypto.outpayload -> merge.payload
flow merge.frame -> outframe
```
The Slicing Approach

Slicing the Crypto Controller

```plaintext
system cryptocontroller(
    inframe: in data (int,int)
    outframe: out data (int,enc int)
    mykeys: key pair
)

system split(
    frame: in data (int,int)
    header: out data int
    payload: out data int
    m0: initial mode
    m0 -[then header := frame[0];
        payload := frame[1]]-> m0
)

system bypass(
    inheader: in data int
    outheader: out data int
    m0: initial mode
    m0 -[then outheader := inheader]-> m0
)

Add data elements, events and source modes of interesting transitions

system crypto(
    inpayload: in data int 0
    outpayload: out data enc int
    k: key pub(mykeys)
    m0: initial mode
    m0 -[then outpayload := encrypt(inpayload,k)]-> m0
)

system merge(
    header: in data int
    payload: in data enc int
    frame: out data (int,enc int)
    m0: initial mode
    m0 -[then frame := (header,payload)]-> m0
)

flow inframe -> split.frame
flow split.header -> bypass.inheader
flow split.payload -> crypto.inpayload
flow bypass.outheader -> merge.header
flow crypto.outpayload -> merge.payload
flow merge.frame -> outframe
)
The Slicing Approach

Slicing the Crypto Controller

system cryptocontroller(
    inframe: in data (int,int)
    outframe: out data (int,enc int)
    mykeys: key pair
)

system split(
    frame: in data (int,int)
    header: out data int
    payload: out data int
    m0: initial mode
    m0 -[then header := frame[0]; payload := frame[1]]-> m0
)

system bypass(
    inheader: in data int
    outheader: out data int
    m0: initial mode
    m0 -[then outheader := inheader] -> m0
)

system crypto(
    inpayload: in data int 0
    outpayload: out data enc int
    k: key pub(mykeys)
    m0: initial mode
    m0 -[then outpayload := encrypt(inpayload,k)]-> m0
)

system merge(
    header: in data int
    payload: in data enc int
    frame: out data (int,enc int)
    m0: initial mode
    m0 -[then frame := (header,payload)]-> m0
)

flow inframe -> split.frame
flow split.header -> bypass.inheader
flow split.payload -> crypto.inpayload
flow bypass.outheader -> merge.header
flow crypto.outpayload -> merge.payload
flow merge.frame -> outframe

Add sources and modes of flows with interesting targets
The Slicing Approach

Slicing the Crypto Controller

```plaintext
system cryptocontroller(
    inframe: in data (int,int)
    outframe: out data (int,enc int)
    mykeys: key pair
)

system split(
    frame: in data (int,int)
    header: out data int
    payload: out data int
    m0: initial mode
    m0 -[then header := frame[0];
          payload := frame[1]]-> m0
)

system bypass(
    inheader: in data int
    outheader: out data int
    m0: initial mode
    m0 -[then outheader := inheader]-> m0
)

system crypto(
    inpayload: in data int 0
    outpayload: out data enc int
    k: key pub(mykeys)
    m0: initial mode
    m0 -[then outpayload := encrypt(inpayload,k)]-> m0
)

system merge(
    header: in data int
    payload: in data enc int
    frame: out data (int,enc int)
    m0: initial mode
    m0 -[then frame := (header,payload)]-> m0
)

flow inframe -> split.frame
flow split.header -> bypass.inheader
flow split.payload -> crypto.inpayload
flow bypass.outheader -> merge.header
flow crypto.outpayload -> merge.payload
flow merge.frame -> outframe
```

Add source modes of transitions that affect interesting data elements
The Slicing Approach

Slicing the Crypto Controller

```system cryptocontroller(
    inframe: in data (int,int)
    outframe: out data (int,enc int)
    mykeys: key pair
)

system split(
    frame: in data (int,int)
    header: out data int
    payload: out data int
    m0: initial mode
    m0 -[then header := frame[0]; payload := frame[1]]-> m0
)

system bypass(
    inheader: in data int
    outheader: out data int
    m0: initial mode
    m0 -[then outheader := inheader]-> m0
)

Add data elements, events and source modes of interesting transitions

```system crypto(
    inpayload: in data int 0
    outpayload: out data enc int
    k: key pub(mykeys)
    m0: initial mode
    m0 -[then outpayload := encrypt(inpayload,k)]-> m0
)

system merge(
    header: in data int
    payload: in data enc int
    frame: out data (int,enc int)
    m0: initial mode
    m0 -[then frame := (header,payload)]-> m0
)

flow inframe -> split.frame
flow split.header -> bypass.inheader
flow split.payload -> crypto.inpayload
flow bypass.outheader -> merge.header
flow crypto.outpayload -> merge.payload
flow merge.frame -> outframe
```
The Slicing Approach

Slicing the Crypto Controller

system cryptocontroller(
    inframe: in data (int,int)
   outframe: out data (int,enc int)
   mykeys: key pair
)

system split(
    frame: in data (int,int)
    header: out data int
    payload: out data int
    m0: initial mode
    m0 -[then header := frame[0];
         payload := frame[1]]-> m0
)

system bypass(
    inheader: in data int
    outheader: out data int
    m0: initial mode
    m0 -[then outheader := inheader]-> m0
)

Add sources and modes of flows with interesting targets

system crypto(
    inpayload: in data int 0
    outpayload: out data enc int
    k: key pub(mykeys)
    m0: initial mode
    m0 -[then outpayload := encrypt(inpayload,k)]-> m0
)

system merge(
    header: in data int
    payload: in data enc int
    frame: out data (int,enc int)
    m0: initial mode
    m0 -[then frame := (header,payload)]-> m0
)

flow inframe -> split.frame
flow split.header -> bypass.inheader
flow split.payload -> crypto.inpayload
flow bypass.outheader -> merge.header
flow crypto.outpayload -> merge.payload
flow merge.frame -> outframe

20 of 27 Analysing Cryptographically-Masked Information Flows in MILS-AADL Specifications
Thomas Noll
MOVES Söllerhaus Workshop; January 22–27, 2017
The Slicing Approach

Slicing the Crypto Controller

system cryptocontroller(
inframe: in data (int,int)
outframe: out data (int,enc int)
mykeys: key pair
)

system split(
frame: in data (int,int)
header: out data int
payload: out data int
m0: initial mode
m0 -[then header := frame[0]; payload := frame[1]]-> m0
)

system bypass(
inheader: in data int
outheader: out data int
m0: initial mode
m0 -[then outheader := inheader]-> m0
)

system crypto(
inpayload: in data int 0
outpayload: out data enc int
k: key pub(mykeys)
m0: initial mode
m0 -[then outpayload := encrypt(inpayload,k)]-> m0
)

system merge(
header: in data int
payload: in data enc int
frame: out data (int,enc int)
m0: initial mode
m0 -[then frame := (header,payload)]-> m0
)

flow inframe -> split.frame
flow split.header -> bypass.inheader
flow split.payload -> crypto.inpayload
flow bypass.outheader -> merge.header
flow crypto.outpayload -> merge.payload
flow merge.frame -> outframe

Add source modes of transitions that affect interesting data elements
**The Slicing Approach**

**Slicing the Crypto Controller**

system cryptocontroller(
  inframe: in data (int,int)
  outframe: out data (int,enc int)
  mykeys: key pair
)

system split(
  frame: in data (int,int)
  header: out data int
  payload: out data int
  m0: initial mode
  m0 -[then header := frame[0]; payload := frame[1]]-> m0
)

system bypass(
  inheader: in data int
  outheader: out data int
  m0: initial mode
  m0 -[then outheader := inheader]-> m0
)

system crypto(
  inpayload: in data int 0
  outpayload: out data enc int
  k: key pub(mykeys)
  m0: initial mode
  m0 -[then outpayload := encrypt(inpayload,k)]-> m0
)

system merge(
  header: in data int
  payload: in data enc int
  frame: out data (int,enc int)
  m0: initial mode
  m0 -[then frame := (header,payload)]-> m0
)

flow inframe -> split.frame
flow split.header -> bypass.inheader
flow split.payload -> crypto.inpayload
flow bypass.outheader -> merge.header
flow crypto.outpayload -> merge.payload
flow merge.frame -> outframe

Add data elements, events and source modes of interesting transitions
The Slicing Approach

Slicing the Crypto Controller

system cryptocontroller(
   inframe: in data (int,int)
   outframe: out data (int,enc int)
   mykeys: key pair
)

system split(
   frame: in data (int,int)
   header: out data int
   payload: out data int
   m0: initial mode
   m0 ->[then header := frame[0];
      payload := frame[1]]-> m0
)

system bypass(
   inheader: in data int
   outheader: out data int
   m0: initial mode
   m0 ->[then outheader := inheader]-> m0
)

Add sources and modes of flows with interesting targets

flow inframe -> split.frame
flow split.header -> bypass.inheader
flow split.payload -> crypto.inpayload
flow bypass.outheader -> merge.header
flow crypto.outpayload -> merge.payload
flow merge.frame -> outframe

system crypto(
   inpayload: in data int 0
   outpayload: out data enc int
   k: key pub(mykeys)
   m0: initial mode
   m0 ->[then outpayload := encrypt(inpayload,k)]-> m0
)

system merge(
   header: in data int
   payload: in data enc int
   frame: out data (int,enc int)
   m0: initial mode
   m0 ->[then frame := (header,payload)]-> m0
)
The Slicing Approach

Slicing the Crypto Controller

```plaintext
system cryptocontroller(
    inframe: in data (int,int)
    outframe: out data (int,enc int)
    mykeys: key pair
)

system split(
    frame: in data (int,int)
    header: out data int
    payload: out data int
    m0: initial mode
    m0 - [ then header := frame[0];
            payload := frame[1] ] -> m0
)

system bypass(
    inheader: in data int
    outheader: out data int
    m0: initial mode
    m0 - [ then outheader := inheader ] -> m0
)

system crypto(
    inpayload: in data int 0
    outpayload: out data enc int
    k: key pub(mykeys)
    m0: initial mode
    m0 - [ then outpayload := encrypt(inpayload,k) ] -> m0
)

system merge(
    header: in data int
    payload: in data enc int
    frame: out data (int,enc int)
    m0: initial mode
    m0 - [ then frame := (header,payload) ] -> m0
)

flow inframe -> split.frame
flow split.header -> bypass.inheader
flow split.payload -> crypto.inpayload
flow bypass.outheader -> merge.header
flow crypto.outpayload -> merge.payload
flow merge.frame -> outframe
```

Thus: (low) outframe depends on (high)

inframe \(\Rightarrow\) (classical) interference!
Handling Cryptographic Operations

Content

Context: The D-MILS Project

Information Flow Security

The Slicing Approach

Handling Cryptographic Operations
Handling Cryptographic Operations

Handling Encryption and Decryption

Security concepts in MILS-AADL

- Declaration of key pairs as global constants on top level ("mykeys: key pair")
- Assignment of (public/private) subkeys to data subcomponents ("k: key pub(mykeys")
- Forwarding via data ports possible

⇒ Static pool of keys with dynamic distribution
Handling Cryptographic Operations

Handling Encryption and Decryption

Security concepts in MILS-AADL

- Declaration of key pairs as global constants on top level ("mykeys: key pair")
- Assignment of (public/private) subkeys to data subcomponents ("k: key pub(mykeys)")
- Forwarding via data ports possible
  ⇒ Static pool of keys with dynamic distribution

Assumptions

- Attackers cannot distinguish between low-equivalent ciphertexts
- Decryption using wrong key fails
Handling Cryptographic Operations

Handling Encryption and Decryption

Security concepts in MILS-AADL

- Declaration of key pairs as **global constants** on top level (“mykeys: key pair”)
- Assignment of (public/private) **subkeys** to data subcomponents (“k: key pub(mykeys)”)  
- **Forwarding** via data ports possible  
  ⇒ **Static pool** of keys with **dynamic distribution**

Assumptions

- Attackers cannot distinguish between low-equivalent ciphertexts
- Decryption using wrong key fails

Challenges

- Detect “illegal” releases of keys
- Evaluate security level of encrypted/decrypted information
Handling Cryptographic Operations

Conditional Slicing

Analysis approach

Conditional slicing of specification w.r.t. knowledge of (private) keys
Handling Cryptographic Operations

Conditional Slicing

Analysis approach

Conditional slicing of specification w.r.t. knowledge of (private) keys

Encryption: \( \text{encrypt}(\text{val}, \text{key}) \)

- maintain sets of data elements \( (C) \) and public keys \( (U) \) that \textit{may} contribute to first/be used as second argument
- result always \textbf{declassified} to \( L \)
Handling Cryptographic Operations

Conditional Slicing

Analysis approach

Conditional slicing of specification w.r.t. knowledge of (private) keys

Encryption: \texttt{encrypt(val, key)}

- maintain sets of data elements \((C)\) and public keys \((U)\) that \textit{may} contribute to first/be used as second argument
- result always \texttt{declassified} to \(L\)

Decryption: \texttt{decrypt(val, key)}

- maintain sets of \((C, U)\)-pairs and private keys \((P)\) that \textit{may} contribute to first/be used as second argument
- result depends on \(C' = \bigcup \{C \mid U \cap P \neq \emptyset\}\)
- re-classification: resulting security level is maximal level in \(C'\)
Handling Cryptographic Operations

Conditional Slicing Algorithm

\[ D := S; \quad CU := \emptyset; \quad E := \emptyset; \quad M := \emptyset; \quad /* Input: slicing criterion \( S \subseteq Dat \) */ \]

\begin{verbatim}
repeat
  for all \( m \xrightarrow{e,g,f} m' \in Trn \) with \( \exists d \in D : f \) updates \( d \) or \( \exists d \in D : d \) inactive in \( m \) but active in \( m' \) or \( e \in E \) do
    \( M := M \cup \{ m \} \);
  for all \( m \xrightarrow{e,g,f} m' \in Trn \) with \( m \in M \) or \( m' \in M \) do
    \( D := D \cup \{ d \in Dat \mid g \) reads \( d \}; \quad E := E \cup \{ e \}; \quad M := M \cup \{ m \}; \)
    for all \( d' := a \) in \( f \) with \( d' \in D \) do
      if \( a = encrypt(d, k) \) then
        \( (C, CU', E', M') := \text{CondSlice}\{d\}; (U, CU'', E'', M'') := \text{CondSlice}\{k\}; \)
        \( CU := CU \cup \{(C, U)\}; \quad E := E \cup E' \cup E''; \quad M := M \cup M' \cup M''; \)
      else if \( a = decrypt(d, k) \) then
        \( (D', CU', E', M') := \text{CondSlice}\{d\}; (P, CU'', E'', M'') := \text{CondSlice}\{k\}; \)
        \( D := D \cup \{ C \mid (C, U) \in CU', U \cap P \neq \emptyset \}; \quad E := E \cup E' \cup E''; \quad M := M \cup M' \cup M''; \)
      else
        \( D := D \cup \{ d \mid a \) reads \( d \}; \)
    for all \( a \sim d' \in Flw \) with \( d \in D \) do
      \( D := D \cup \{ d \in Dat \mid a \) reads \( d \}; \quad M := M \cup \{ m \in Mod \mid a \sim d' \) active in \( m \}; \)
    for all \( e \sim e' \in Con \) with \( e \in E \) or \( e' \in E \) do
      \( E := E \cup \{ e, e' \}; \quad M := M \cup \{ m \in Mod \mid e \sim e' \) active in \( m \}; \)
  until nothing changes;
return \((D, CU, E, M)\)
\end{verbatim}
Handling Cryptographic Operations

Example: Secure Communication

1. In crypto:
   \[ \text{outpayload} := \text{encrypt}(\text{inpayload}, k_1) \]
   with \[ k_1 = \text{pub}(\text{mykeys}) \]
   \[ C = \{ \text{split}_1.\text{payload}, \text{split}_1.\text{frame}, \text{inframe} \} \]

2. In decrypto:
   \[ \text{outpayload} := \text{decrypt}(\text{inpayload}, k_2) \]
   with \[ k_2 = \text{priv}(\text{mykeys}) \]
   \[ P = \{ \text{mykeys} \} \]
   \[ P \cap U = \{ \text{mykeys} \} \neq \emptyset \]
   \[ C = \{ \text{split}_1.\text{payload}, \text{split}_1.\text{frame}, \text{inframe} \} \] added to \( D \).
Handling Cryptographic Operations

Example: Secure Communication

1. In crypto: outpayload := encrypt(inpayload, k1) with k1 = pub(mykeys)
   - C = {split₁.payload, split₁.frame, inframe}
   - U = {mykeys}
Handling Cryptographic Operations

Example: Secure Communication

1. In crypto: \( \text{outpayload} := \text{encrypt}(\text{inpayload}, k_1) \) with \( k_1 = \text{pub(} \text{mykeys}) \)
   - \( C = \{\text{split}_1.\text{payload}, \text{split}_1.\text{frame}, \text{inframe}\} \)
   - \( U = \{\text{mykeys}\} \)

2. In decrypto: \( \text{outpayload} := \text{decrypt}(\text{inpayload}, k_2) \) with \( k_2 = \text{priv(} \text{mykeys}) \)
   - \( P = \{\text{mykeys}\} \)
   \( P \cap U = \{\text{mykeys}\} \neq \emptyset \)
   \( C = \{\text{split}_1.\text{payload}, \text{split}_1.\text{frame}, \text{inframe}\} \) added to \( D \)

\[ (L, H) \]

\[ \text{split}_1 \rightarrow \text{bypass}_1 \rightarrow \text{crypto} \rightarrow \text{merge}_1 \rightarrow (L, L) \]

\[ \text{split}_2 \rightarrow \text{merge}_2 \rightarrow \text{bypass}_2 \rightarrow \text{decrypto} \rightarrow (L, H) \]
Theorem

Let

- \( \sigma : \text{Dat} \cup \text{Evt} \cup \text{Mod} \rightarrow \{\text{L}, \text{H}\} \) a security assignment,
- \( S \subseteq \text{Dat} \) with \( \sigma(S) = \{\text{L}\} \), and
- \((D, CU, E, M)\) the backward slice of \( S \).

If

\[ \sigma(D) \cup \sigma(E) \cup \sigma(M) = \{\text{L}\}, \]

then \( \sigma \) ensures possibilistic non-interference.
Handling Cryptographic Operations

Conditional Slicing and Possibilistic Non-Interference

**Theorem**

Let

- \( \sigma : Dat \cup Evt \cup Mod \rightarrow \{L, H\} \) a security assignment,
- \( S \subseteq Dat \) with \( \sigma(S) = \{L\} \), and
- \((D, CU, E, M)\) the backward slice of \( S \).

If

\[ \sigma(D) \cup \sigma(E) \cup \sigma(M) = \{L\}, \]

then \( \sigma \) ensures **possibilistic non-interference**.

**To be done**

Inference rather than checking of security assignments (using constraint solving)
The End

Questions?