Analysing Cryptographically-Masked Information Flows in D-MILS Architectures
— Preliminary Results —

Thomas Noll (noll@cs.rwth-aachen.de)
MOVES Söllerhaus Workshop; March 4, 2015
The D-MILS Project

Content

The D-MILS Project

Information Flow Security

The Type Checking Approach

The Slicing Approach
Distributed MILS

- Funded by the 7th Framework Programme of the European Commission
- Website d-mils.org
- Project consortium
D-MiLS Design Flow

Architectural Refinement

Security Analysis → MILS AADL → MILS AADL → MILS AADL → Safety Analysis

Performance Analysis

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

Implements/Satisfies

Configuration Compiler

Autofocus Model
C Code
Autocode

Simulink Model
C Code
Autocode

Ada Code

Compiler
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.

This component has no interaction with any other.

Suitability of the architecture for some purpose presumes that the architect’s assumptions are met in the implementation of the architecture diagram.
MILS Platform – Provides Straightforward Realization of Policy Architecture

Architecture
Validity of the architecture assumes that the only interactions of the circles (operational components) is through the arrows depicted in the diagram

Realization
SK, with other MILS foundational components, form the MILS Platform allowing operational components to share physical resources while enforcing Isolation and Information Flow Control
Distributed MILS (D-MILS): Policy architecture deployment spanning nodes

MNS – MILS Networking System    SK – Separation Kernel
D-MILS Research and Technology Development Areas

Architecture
Analysis and Design
Language
MILS-AADL

Intermediate Languages

Verification Framework

Assurance Framework

MILS Platform Configuration Compiler

Target Configuration tools

D-MILS Platform target

LSK TTE

Graphical & Declarative Languages

Compositional Verification

Compositional Assurance Case

Representation Semantics and Transformations

Integration GSN & AADL

Configuration Synthesis

D-MILS Platform

Pre-existing products

Behavior Annotation

Property Annotation

Goal Structuring Notation

D-MILS Project Overview © 2015 D-MILS Project
Information Flow Security

Content

The D-MILS Project

Information Flow Security

The Type Checking Approach

The Slicing Approach
Information flow security

Non-interference: “High-security inputs have no effects on low-security outputs”

- Non-interference property includes:
  - Confidentiality (secrets kept)
  - Integrity (data not corrupted)
Information Flow Security

Some Security Concepts

- 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

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

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

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

Interpretation: behaviour seen by "low" observer unaffected by changes in "high" behaviour
Information Flow Security

Some Security Concepts

- 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 ($t_1 \sim_ET_2$) 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 $t_1, t_2 \in T$,

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

Interpretation: behaviour seen by “low” observer unaffected by changes in “high” behaviour
Information Flow Security

Some Security Concepts

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

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

Interpretation: behaviour seen by “low” observer unaffected by changes in “high” behaviour
Cryptographically-Masked Information Flow

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

Example (Password encryption)

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

\[ t_1 \mid \text{In} \cap s^{-1}(L) = \varepsilon = t_2 \mid \text{In} \cap s^{-1}(L), \text{but} \quad t_1 \mid \text{Out} \cap s^{-1}(L) \neq t_2 \mid \text{Out} \cap s^{-1}(L) \]

⇒ Interference

Common approach: declassification
- Allows security level of incoming information to be lowered (here: password)
- Categorisation according to where/who/when/what [Sabelfeld/Sands 2005]

Problems:
- exceptions to security policy might introduce unforeseen information release
- systematic handling of re-classification unclear
Information Flow Security

Cryptographically-Masked Information Flow

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

Example (Password encryption)

- $In = \{pwd_1_H, pwd_2_H\}$, $Out = \{enc_1_L, enc_2_L\}$
- $t_1 = pwd_1 \cdot enc_1$, $t_2 = pwd_2 \cdot enc_2$
- $t_1|_{In \cap s^{-1}(L)} = \varepsilon = t_2|_{In \cap s^{-1}(L)}$, but $t_1|_{Out \cap s^{-1}(L)} = enc_1 \neq enc_2 = t_2|_{Out \cap s^{-1}(L)}$

$\Rightarrow$ Interference
Information Flow Security

Cryptographically-Masked Information Flow

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

Example (Password encryption)

- $In = \{pwd_1^H, pwd_2^H\}$, $Out = \{enc_1^L, enc_2^L\}$
- $t_1 = pwd_1 \cdot enc_1$, $t_2 = pwd_2 \cdot enc_2$
- $t_1|_{In \cap s^{-1}(L)} = \varepsilon = t_2|_{In \cap s^{-1}(L)}$, but $t_1|_{Out \cap s^{-1}(L)} = enc_1 \neq enc_2 = t_2|_{Out \cap s^{-1}(L)}$
- $\Rightarrow$ Interference

Common approach: declassification

- Allows security level of incoming information to be lowered (here: password)
- Categorisation according to *where/who/when/what* [Sabelfeld/Sands 2005]
- Problems:
  - exceptions to security policy might introduce unforeseen information release
  - systematic handling of re-classification unclear
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 may not distinguish between ciphertexts
- Naive approach: all ciphertexts are indistinguishable
- But: enables occlusion (i.e., security leaks by implicit data flow)
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 may not distinguish between ciphertexts.
- Naive approach: all ciphertexts are indistinguishable.
- But: enables occlusion (i.e., security leaks by implicit data flow).

**Example (Occlusion)**

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

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 may not distinguish between ciphertexts
- Naive approach: all ciphertexts are indistinguishable
- But: enables occlusion (i.e., security leaks by implicit data flow)

Example (Occlusion)

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

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 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 if sets of possible results coincide

Definition

\( \sim \) is a low-equivalence relation on ciphertexts if \( \forall v_1, v_2, k_1, k_2: \)

1. safe usage: \( \forall u_1 \in \text{encrypt}(v_1, k_1). \exists u_2 \in \text{encrypt}(v_2, k_2) : u_1 \sim u_2 \)
2. prevent occlusion: \( \exists u_1 \in \text{encrypt}(v_1, k_1), u_2 \in \text{encrypt}(v_2, k_2) : u_1 \not\sim u_2 \)
Information Flow Security

Possibilistic Non-Interference [McCullough 1988]

- Encryption non-deterministically calculates a ciphertext out of a set
- Encrypted values low-equivalent if sets of possible results coincide

Definition

\[ \sim_L \] is a low-equivalence relation on ciphertexts if \( \forall v_1, v_2, k_1, k_2: \)

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. prevent 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 values and environments

Definition (Possibilistic non-interference (informal))

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.
Example (Safe usage of encryption)

\[
\text{m0} \rightarrow \text{then low := encrypt(high, key)} \rightarrow \text{m1};
\]

- 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
  1. \( E'_1 = \{ \eta_1[\text{low} \mapsto u_1] \mid u_1 \in \text{encrypt}(v_1, k) \} \)
  2. \( E'_2 = \{ \eta_2[\text{low} \mapsto u_2] \mid u_2 \in \text{encrypt}(v_2, k) \} \)
- Now \( \forall u_1 \in \text{encrypt}(v_1, k_1). \exists u_2 \in \text{encrypt}(v_2, k_2) : u_1 \sim_L u_2 \) 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)

\[
m_0 \rightarrow \text{then } \text{low}_1 := \text{encrypt}(\text{val}, \text{key}) \rightarrow m_1;
m_1 \rightarrow \text{when high then } \text{low}_2 := \text{encrypt}(\text{val}, \text{key}) \rightarrow m_2;
m_1 \rightarrow \text{when not high then } \text{low}_2 := \text{low}_1 \rightarrow m_2;
\]

- Let \( \sigma(\text{high}) = \sigma(\text{val}) = H \) and \( \sigma(\text{key}) = \sigma(\text{low}_1) = \sigma(\text{low}_2) = L \)
- Let environments \( \eta_1, \eta_2 \) with \( \eta_1 \sim_L \eta_2 \) such that
  1. \( \eta_1(\text{high}) = true, \eta_1(\text{val}) = \nu_1, \eta_1(\text{key}) = k \)
  2. \( \eta_2(\text{high}) = false, \eta_2(\text{val}) = \nu_2, \eta_2(\text{key}) = k \)
- Execution respectively yields
  1. \( E'_1 = \{ \eta_1[\text{low}_1 \mapsto u_1, \text{low}_2 \mapsto u_2] \mid u_1 \in \text{encrypt}(\nu_1, k), u_2 \in \text{encrypt}(\nu_2, k) \} \)
  2. \( E'_2 = \{ \eta_2[\text{low}_1 \mapsto u, \text{low}_2 \mapsto u] \mid u \in \text{encrypt}(\nu_1, k) \} \)
- Now \( \exists u_1 \in \text{encrypt}(\nu_1, k), u_2 \in \text{encrypt}(\nu_2, k) : u_1 \not\sim_L u_2 \) implies that
  \( \exists \eta'_1 \in E'_1 : \eta'_1(\text{low}_1) \not\sim_L \eta'_1(\text{low}_2) \)
- On the other hand, \( \forall \eta'_2 \in E'_2 : \eta'_2(\text{low}_1) \sim_L \eta'_2(\text{low}_2) \)
- Thus \( \exists \eta'_1 \in E'_1. \forall \eta'_2 \in E'_2 : \eta'_1 \not\sim_L \eta'_2 \)
- \( \Rightarrow \) Possibilistic interference
The Type Checking Approach

Content

The D-MILS Project

Information Flow Security

The Type Checking Approach

The Slicing Approach
The Type Checking Approach

MILS-AADL Specifications

crypto controller

split

bypass

crypto

merge
The Type Checking Approach

- Introduce typing environment $T$
  - local variables and data ports $\rightarrow$ security type $\tau$ (data type $t$ + security level $\sigma$)
  - modes and event ports $\rightarrow$ security level $\sigma$

Theorem ([MILS Workshop 2015]): If the system is typeable, it is possibilistically non-interfering.
The Type Checking Approach

The Type Checking Approach

- Introduce typing environment $T$
  - local variables and data ports $\rightarrow$ security type $\tau$ (data type $t +$ security level $\sigma$)
  - modes and event ports $\rightarrow$ security level $\sigma$
- Specify typing rules
  - parametrised by $T$
  - derive types of connections and transitions
- Example: encryption and decryption

\[
\begin{align*}
T \vdash e_1 : \tau & \quad T \vdash e_2 : \text{key } L \\
T \vdash \text{encrypt}(e_1, e_2) : \text{enc } \tau L
\end{align*}
\]

\[
\begin{align*}
T \vdash e_1 : \text{enc } \tau \sigma & \quad T \vdash e_2 : \text{key } H \\
T \vdash \text{decrypt}(e_1, e_2) : \tau^\sigma
\end{align*}
\]

Theorem ([MILS Workshop 2015])

If the system is typeable, it is possibilistically non-interfering.
The Type Checking Approach

• Introduce typing environment $T$
  – local variables and data ports → security type $\tau$ (data type $t$ + security level $\sigma$)
  – modes and event ports → security level $\sigma$

• Specify typing rules
  – parametrised by $T$
  – derive types of connections and transitions

• Example: encryption and decryption

$$
T \vdash e_1 : \tau \quad T \vdash e_2 : \text{key } L
$$

$$
T \vdash \text{encrypt}(e_1, e_2) : \text{enc } \tau L
$$

$$
T \vdash e_1 : \text{enc } \tau \sigma \quad T \vdash e_2 : \text{key } H
$$

$$
T \vdash \text{decrypt}(e_1, e_2) : \tau^\sigma
$$

Theorem ([MILS Workshop 2015])

If the system is typeable, it is possibilistically non-interfering.
The Type Checking Approach

Ongoing Work

- Exact characterisation of determinism requirements
  - non-interference property is non-compositional in presence of non-determinism
- Elaboration of correctness proof for type system
- Improving usability by type inference (rather than type checking)
  - based on given security-level assignment to (some) event and data ports
- Implementation of type checking/inference
The Slicing Approach

Content

The D-MILS Project

Information Flow Security

The Type Checking Approach

The Slicing Approach
The Slicing Approach

Motivation

Weaknesses of type checking approach:

- Analysis is flow-insensitive

Example

```
 m0 -[when high then low := 42]-\rightarrow m1;
 m1 -[then low := 0]-\rightarrow m2;

- choosing \(\sigma(low) = L\) is ok since m0 transition has "dead" effect
- but type system cannot handle this (as types are global)
```
The Slicing Approach

Motivation

Weaknesses of type checking approach:

- Analysis is flow-insensitive

Example

m0 - [when high then low := 42] -> m1;
m1 - [then low := 0] -> m2;

- choosing $\sigma(\text{low}) = \text{L}$ is ok since m0 transition has “dead” effect
- but type system cannot handle this (as types are global)

- Analysis does not take (non-)knowledge of encryption keys into account:

$$
\frac{\Gamma \vdash e_1 : \text{enc}(\text{int H}) \; \text{L} \quad \Gamma \vdash e_2 : \text{key} \; \text{H}}{\Gamma \vdash \text{decrypt}(e_1, e_2) : \text{int H}}
$$

yields $\sigma(\text{decrypt}(e_1, e_2)) = \text{H}$ even if $e_2$ cannot be the matching private key
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
- analysis inherently flow-sensitive!
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
- analysis inherently flow-sensitive!

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 [NFM 2010]

\[
D := S; E := \emptyset; M := \emptyset; \} \text{ Initialization based on slicing criterion } S (= \text{ subset of data elements)} \\
\text{repeat} \\
\text{for all } m, g, f \rightarrow m' \in \text{Trn} \text{ with } \exists d \in D: f \text{ updates } d \text{ or } \exists d \in D: d \text{ inactive in } m \text{ but active in } m' \text{ or } e \in E \text{ do} \\
\text{M} := \text{M} \cup \{m\} \}
\]

\[
\text{Transitions that affect interesting data elements or have interesting triggers} \\
\text{for all } m, g, f \rightarrow m' \in \text{Trn} \text{ with } m \in \text{M} \text{ or } m' \in \text{M} \text{ do} \\
\text{D} := \text{D} \cup \{d \in \text{Dat} | g \text{ reads } d\} \cup \{d \in \text{Dat} | f \text{ updates some } d' \in \text{D} \text{ reading } d\} \\
\text{E} := \text{E} \cup \{e\} \\
\text{M} := \text{M} \cup \{m \in \text{Mod} | d := a \text{ active in } m\} \}
\]

\[
\text{Transitions from/to interesting modes} \\
\text{for all } a \Rightarrow d \in \text{Flw} \text{ with } d \in D \text{ do} \\
\text{D} := \text{D} \cup \{d' \in \text{Dat} | a \text{ reads } d'\} \\
\text{M} := \text{M} \cup \{m \in \text{Mod} | d := a \text{ active in } m\} \}
\]

\[
\text{Data flows to interesting ports} \\
\text{for all } e \Rightarrow e' \in \text{Con} \text{ with } e \in E \text{ or } e' \in E \text{ do} \\
\text{E} := \text{E} \cup \{e, e'\} \\
\text{M} := \text{M} \cup \{m \in \text{Mod} | e \Rightarrow e' \text{ active in } m\} \}
\]

\[
\text{Connections involving interesting event ports} \\
\text{until nothing changes;}
\]
The Slicing Approach

Slicing AADL Specifications for Model Checking [NFM 2010]

\[ D := S; \quad E := \emptyset; \quad M := \emptyset; \]  
\{ Initialization based on slicing criterion \( S \) (= subset of data elements) \}

repeat
  \begin{align*}
  &\text{for all } m \xrightarrow{e.g.} m' \in \text{Trn} \text{ with } \exists d \in D : f \text{ updates } d \\
  &\quad \text{or } \exists d \in D : d \text{ inactive in } m \text{ but active in } m' \\
  &\quad \text{or } e \in E \text{ do} \\
  &\quad M := M \cup \{m\}; \\
  &\quad \text{Transitions that affect interesting data elements or} \\
  &\quad \text{have interesting triggers} \\
  \end{align*}

until nothing changes;
The Slicing Approach

Slicing AADL Specifications for Model Checking [NFM 2010]

\[ D := S; E := \emptyset; M := \emptyset; \]  
\{ Initialization based on slicing criterion \( S (= \text{subset of data elements}) \) \}

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

\begin{align*}
\text{for all } m \xrightarrow{e,g,f} m' \in \text{Trn} \text{ with } m \in M \text{ or } m' \in M \\
D := D \cup \{ d \in \text{Dat} \mid g \text{ reads } d \} \\
\quad \cup \{ d \in \text{Dat} \mid f \text{ updates some } d' \in D \text{ reading } d \}; \\
E := E \cup \{ e \}; \\
M := M \cup \{ m \};
\end{align*}

Transitions from/to interesting modes

until nothing changes;
The Slicing Approach

Slicing AADL Specifications for Model Checking [NFM 2010]

D := S; E := ∅; M := ∅; } Initialization based on slicing criterion S (= subset of data elements)
repeat
  for all m →́ m′ ∈ Trn with ∃d ∈ D : f updates d
    or ∃d ∈ D : d inactive in m but active in m′
    or e ∈ E do
    M := M ∪ {m};
  Transitions that affect interesting data elements or have interesting triggers
  for all m →́ m′ ∈ Trn with m ∈ M or m′ ∈ M do
    D := D ∪ {d ∈ Dat | g reads d}
    ∪ {d ∈ Dat | f updates some d′ ∈ D reading d};
    E := E ∪ {e};
    M := M ∪ {m};
  Transitions from/to interesting modes
  for all a → d ∈ Flw with d ∈ D do
    D := D ∪ {d′ ∈ Dat | a reads d′};
    M := M ∪ {m ∈ Mod | d := a active in m};
  Data flows to interesting ports
until nothing changes;
The Slicing Approach

Slicing AADL Specifications for Model Checking [NFM 2010]

\[
\begin{align*}
D &:= S; E := \emptyset; M := \emptyset; \} \text{ Initialization based on slicing criterion } S (= \text{subset of data elements}) \\
\text{repeat} & & \text{Transitions that affect interesting data elements or have interesting triggers} \\
\text{for all } m \xrightarrow{e,g,f} m' \in Trn \text{ with } & \exists d \in D : f \text{ updates } d \\
\text{or } & \exists d \in D : d \text{ inactive in } m \text{ but active in } m' \\
\text{or } e \in E \text{ do} & & \text{Transitions from/to interesting modes} \\
M &:= M \cup \{m\}; \\
\text{for all } m \xrightarrow{e,g,f} m' \in Trn \text{ with } m \in M \text{ or } m' \in M \text{ do} & & \text{Data flows to interesting ports} \\
D &:= D \cup \{d \in Dat \mid g \text{ reads } d\} \\
& \cup \{d \in Dat \mid f \text{ updates some } d' \in D \text{ reading } d\}; \\
E &:= E \cup \{e\}; \\
M &:= M \cup \{m\}; \\
\text{for all } a \xrightarrow{} d \in Flw \text{ with } d \in D \text{ do} & & \text{Connections involving interesting event ports} \\
D &:= D \cup \{d' \in Dat \mid a \text{ reads } d'\}; \\
M &:= M \cup \{m \in Mod \mid d := a \text{ active in } m\}; \\
\text{for all } e \xrightarrow{} e' \in Con \text{ with } e \in E \text{ or } e' \in E \text{ do} & & \text{Connections involving interesting event ports} \\
E &:= E \cup \{e, e'\}; \\
M &:= M \cup \{m \in Mod \mid e \xrightarrow{} e' \text{ active in } m\}; \\
\text{until nothing changes;}
\end{align*}
\]
The Slicing Approach

Example: 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

```

25 of 29 Analysing Cryptographically-Masked Information Flows in D-MILS Architectures
Thomas Noll
MOVES Sölterhaus Workshop; March 4, 2015

RWTH AACHEN UNIVERSITY
The Slicing Approach

Example: 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}
The Slicing Approach

Example: 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 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
)
The Slicing Approach

Example: 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 source modes of transitions that affect interesting data elements

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

25 of 29 Analysing Cryptographically-Masked Information Flows in D-MILS Architectures
Thomas Noll
MOVES Söllerhaus Workshop; March 4, 2015
The Slicing Approach

Example: 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 data elements, events and source modes of interesting transitions

25 of 29 Analysing Cryptographically-Masked Information Flows in D-MILS Architectures
Thomas Noll
MOVES Söllerhaus Workshop; March 4, 2015
The Slicing Approach

Example: 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

25 of 29 Analysing Cryptographically-Masked Information Flows in D-MILS Architectures
Thomas Noll
MOVES Söllerhaus Workshop; March 4, 2015
### The Slicing Approach

#### Example: 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

Example: 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

25 of 29 Analysing Cryptographically-Masked Information Flows in D-MILS Architectures
Thomas Noll
MOVES Söllerhaus Workshop; March 4, 2015
The Slicing Approach

Example: 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
)
The Slicing Approach

Example: 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

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

Example: 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

Example: 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 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
)
The Slicing Approach

Example: 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
)

Thus: (low) outframe depends on (high)
    inframe \Rightarrow (classical) interference!

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

25 of 29 Analysing Cryptographically-Masked Information Flows in D-MILS Architectures
Thomas Noll
MOVES Söllerhaus Workshop; March 4, 2015
The Slicing Approach

Handling Encryption and Decryption

- Security concepts in MILS-AADL:
  - declaration of key pairs as global constants on top level (mykeys)
  - assignment of (public/private) subkeys to data subcomponents (k)
  - forwarding via data ports possible

⇒ static pool of keys with dynamic distribution
The Slicing Approach

Handling Encryption and Decryption

- Security concepts in MILS-AADL:
  - declaration of key pairs as global constants on top level (mykeys)
  - assignment of (public/private) subkeys to data subcomponents (k)
  - forwarding via data ports possible
  ⇒ static pool of keys with dynamic distribution

- Analysis approach: conditional slicing w.r.t. knowledge of keys
  - attach security level to each data element (ports and subcomponents)
  - encrypt(val,key):
    - maintain sets of data elements ($D$) and public keys ($U$) that may be used in first/as second argument
    - result depends on all elements of $D$
    - result always declassified to $L$
  - decrypt(val,key):
    - maintain sets of ($D, U$)-pairs and private keys ($P$) that may be used in first/as second argument
    - result depends on $D' = \bigcup \{D \mid U \cap P \neq \emptyset\}$
    - resulting security level is maximal level in $D'$
The Slicing Approach

Example: Secure Communication

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

2. \( \text{outpayload} := \text{decrypt}(\text{inpayload}, k_2) \)
   with \( k_2 = \text{priv(mykeys)} \)
   - \( P = \{ \text{mykeys} \} \)
   \[ \Rightarrow P \cap U = \{ \text{mykeys} \} \neq \emptyset \]
   \[ \Rightarrow D' = \{ \text{split}_1.\text{payload}, \text{split}_1.\text{frame}, \text{inframe} \} \]
The Slicing Approach

Example: Secure Communication

1. outpayload := encrypt(inpayload, k1) with k1 = pub(mykeys)
   - D = {split₁.payload, split₁.frame, inframe}
   - U = {mykeys}
The Slicing Approach

Example: Secure Communication

1. outpayload := encrypt(inpayload, k1) with k1 = pub(mykeys)
   - $D = \{\text{split}_1.\text{payload}, \text{split}_1.\text{frame}, \text{inframe}\}$
   - $U = \{\text{mykeys}\}$

2. outpayload := decrypt(inpayload, k2) with k2 = priv(mykeys)
   - $P = \{\text{mykeys}\}$
   $\Rightarrow P \cap U = \{\text{mykeys}\} \neq \emptyset$
   $\Rightarrow D' = \{\text{split}_1.\text{payload}, \text{split}_1.\text{frame}, \text{inframe}\}$
The Slicing Approach

Ongoing Work

- Work out details of conditional slicing algorithm
- Correctness proof w.r.t. possibilistic non-interference
  - if no low output conditionally depends on any high input, the system is possibilistically non-interfering
- Relation to type checking approach
  - conjecture: if the system is typeable, then no low output conditionally depends on any high input
  - reverse inclusion does not hold due to flow-(in-)sensitivity
The End

Questions?