Analysing Cryptographically-Masked Information Flows in D-MILS Architectures

— Preliminary Results —

Thomas Noll (noll@cs.rwth-aachen.de)

MOVES Söllerhaus Workshop; March 4, 2015

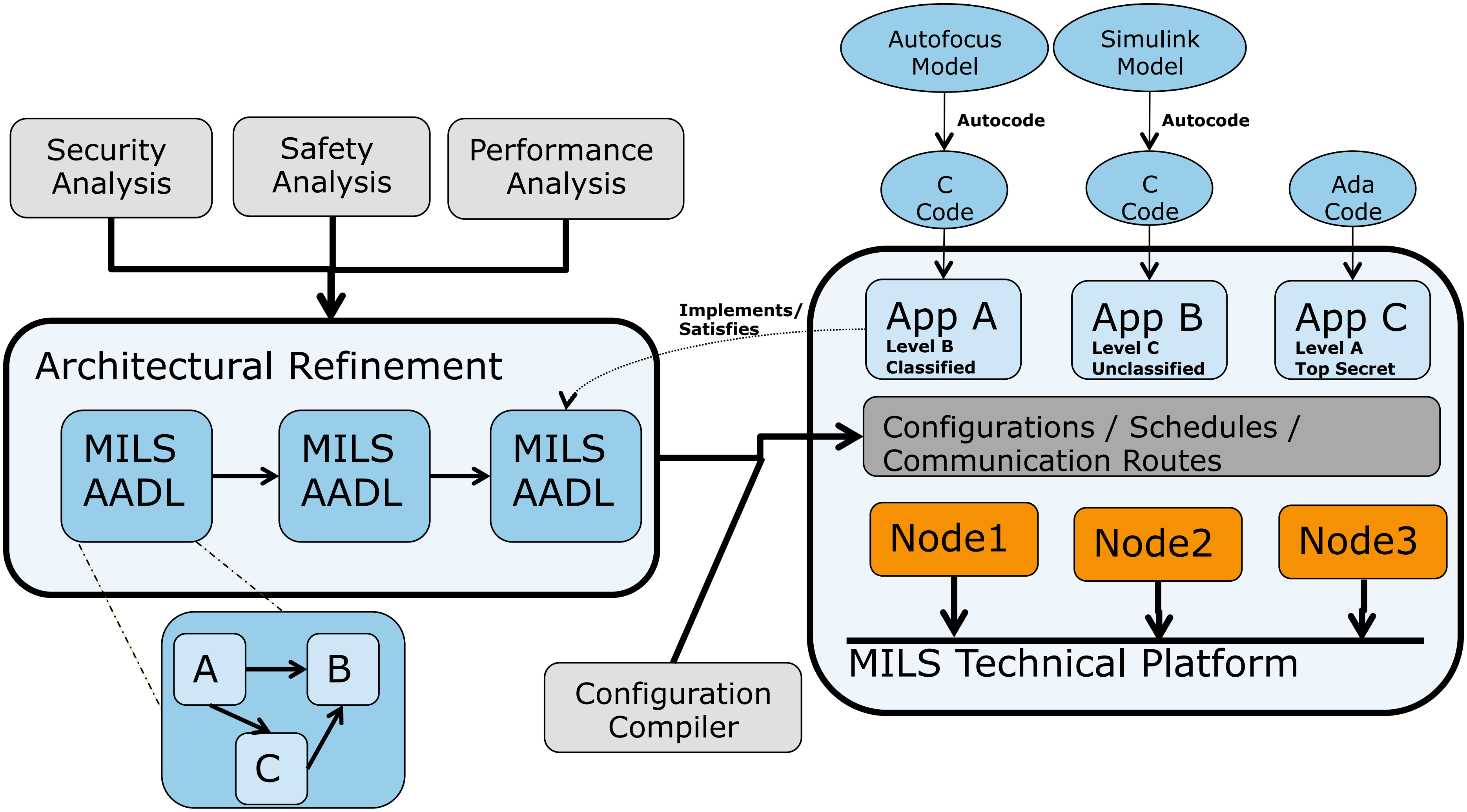
Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

Content

Distributed MILS

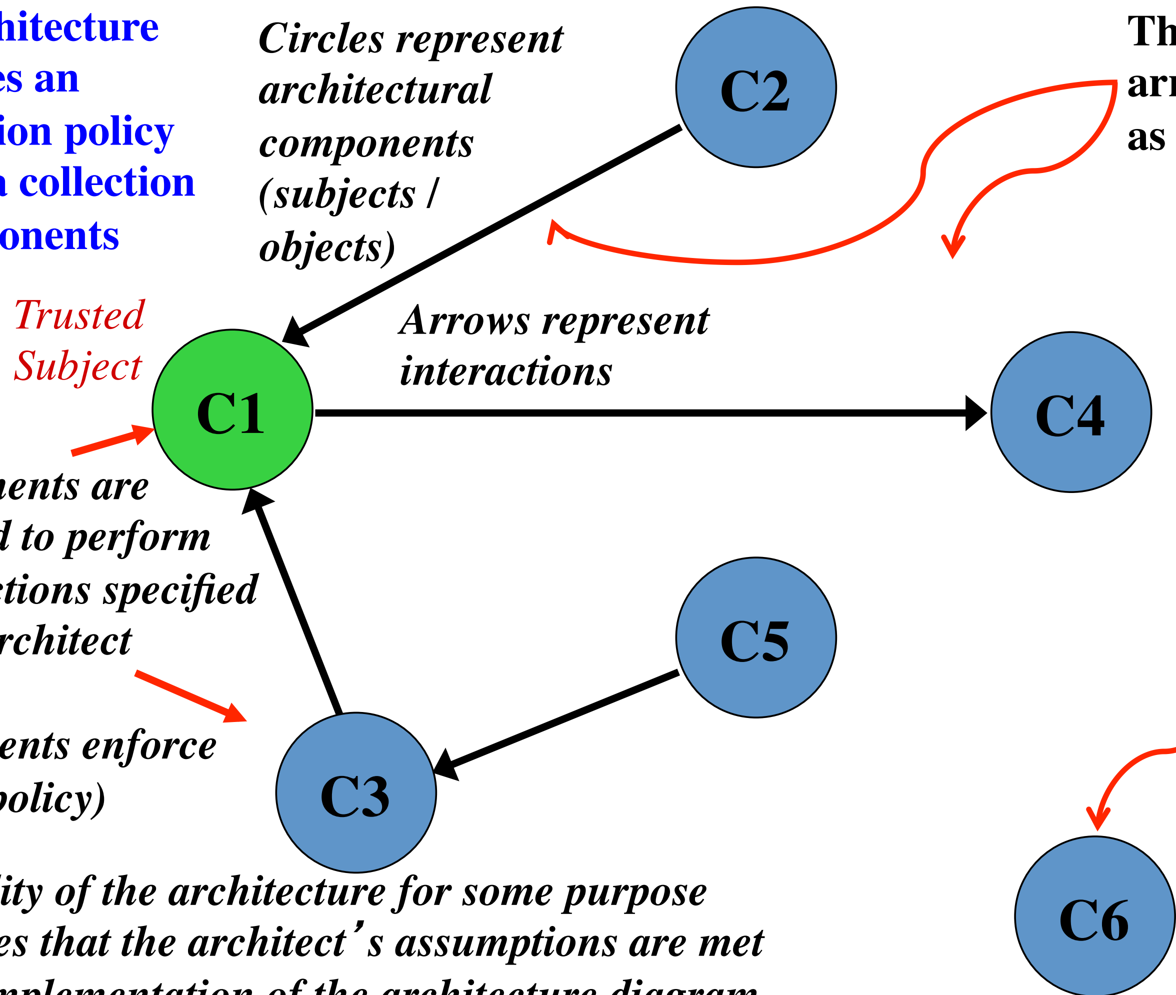
- Funded by the 7th Framework Programme of the European Commission
- Website d-mils.org
- Project consortium

D-MILS Design Flow



MILS Policy Architecture

The architecture expresses an interaction policy among a collection of components



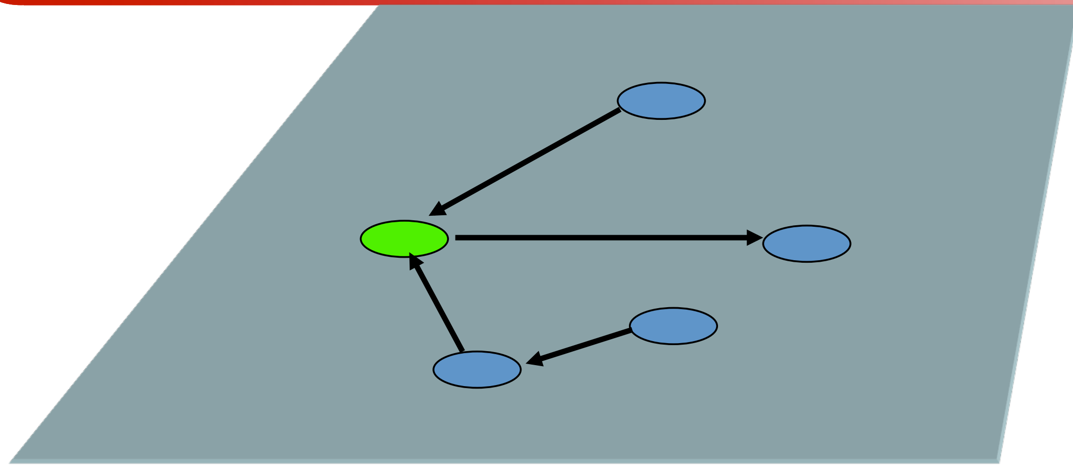
The *absence* of an arrow *is as significant* as the *presence* of one

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

This component has *no interaction* with any other

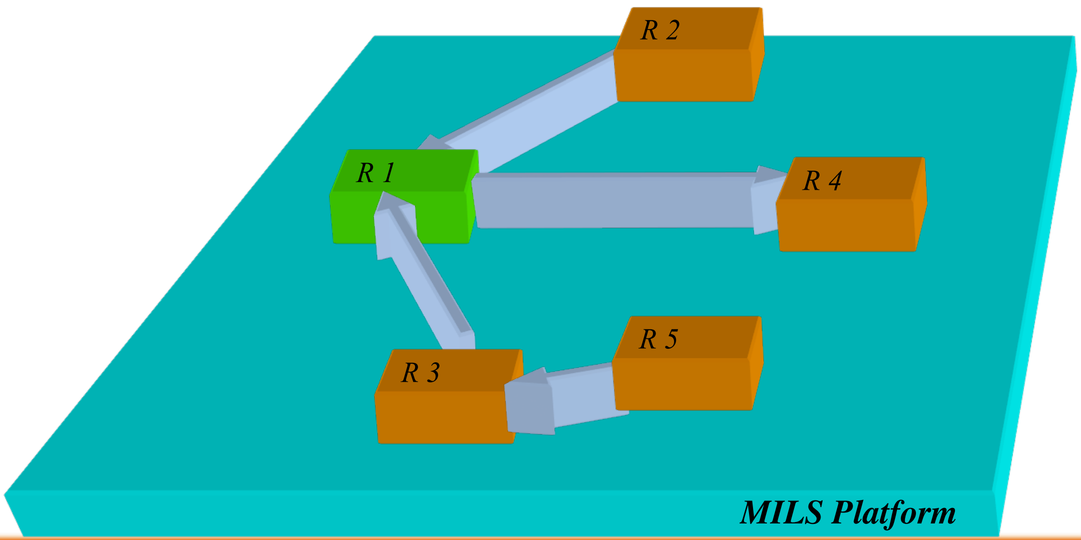
Suitability of the architecture for some purpose presumes that the architect's assumptions are met in the implementation of the architecture diagram.

MILS Platform – Provides Straightforward Realization of Policy Architecture



Architecture

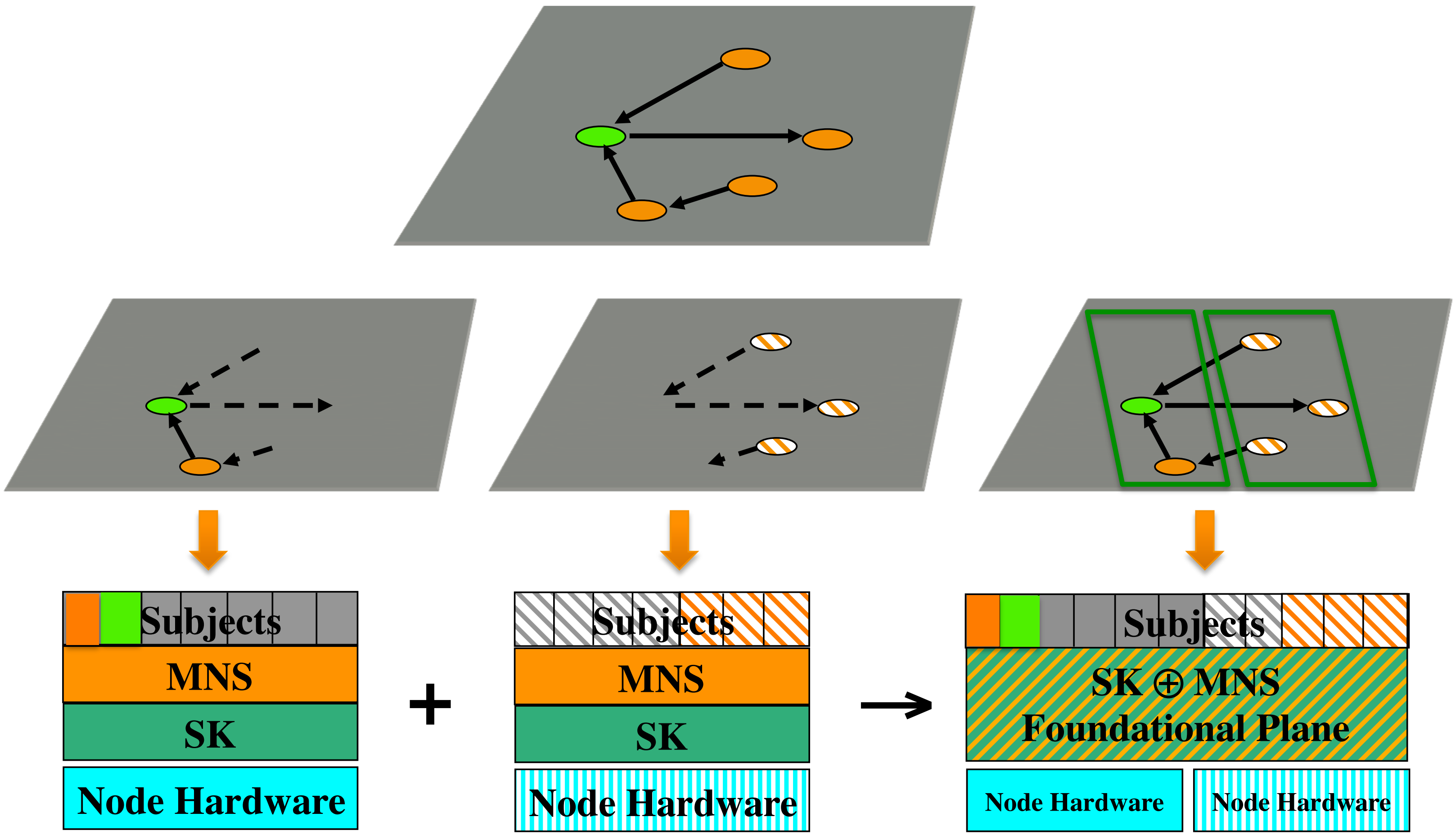
Validity of the architecture assumes that the *only* interactions of the circles (operational components) is through the arrows depicted in the diagram



Realization

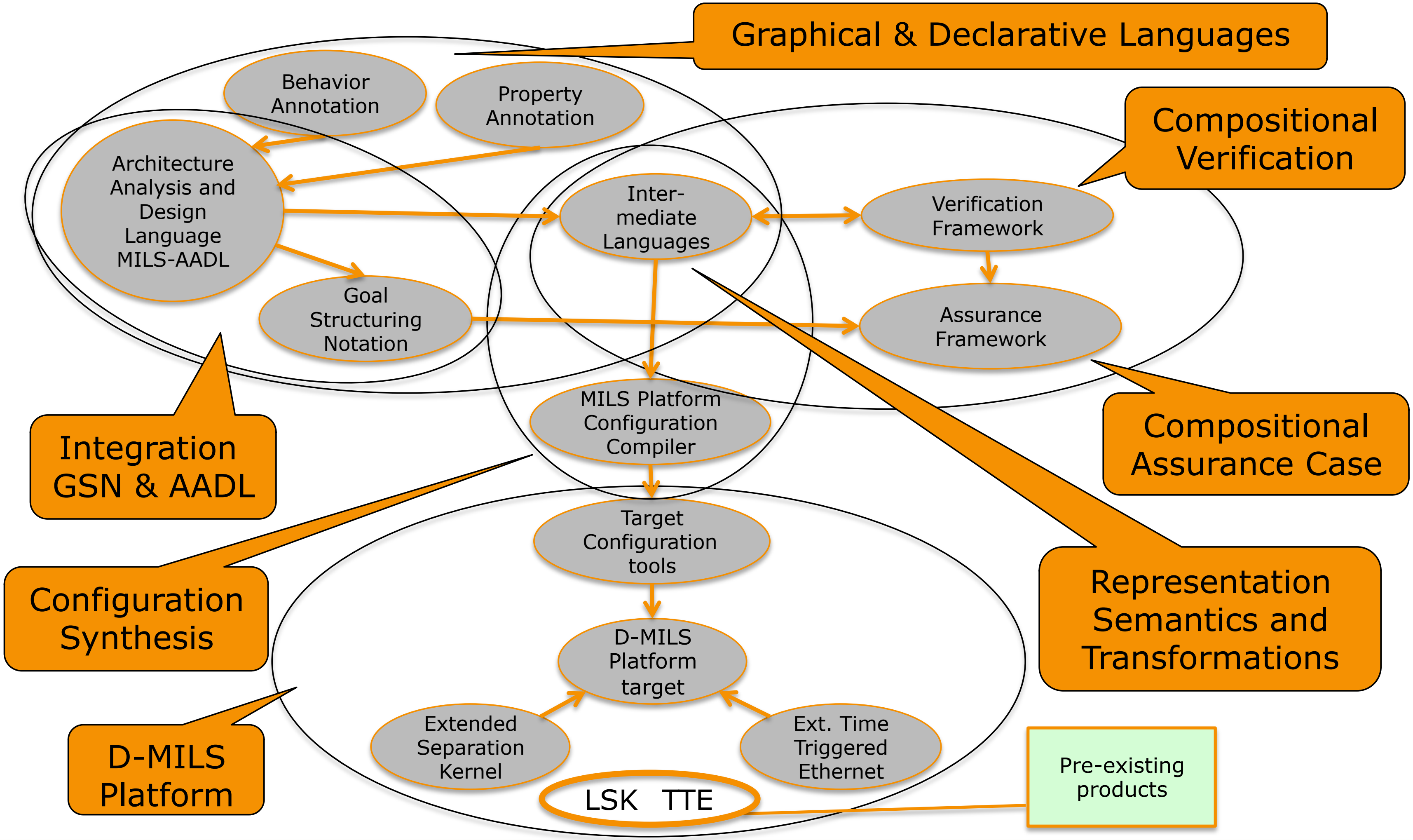
SK, with other MILS foundational components, form the *MILS Platform* allowing operational components to share physical resources while enforcing Isolation and Information Flow Control

Distributed MILS (D-MILS): Policy architecture deployment spanning nodes



MNS – MILS Networking System SK – Separation Kernel

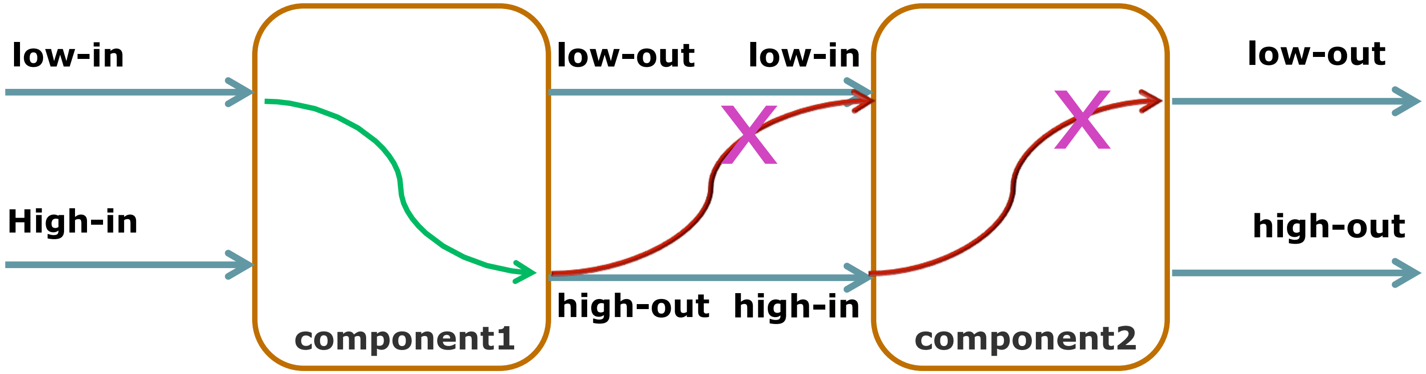
D-MILS Research and Technology Development Areas



Content

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)

Some Security Concepts

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

Some Security Concepts

- Here: two security levels $\mathtt{L}$ (low/public) and $\mathtt{H}$ (high/confidential/secret/private)
  - partial order $\mathtt{L} \sqsubseteq \mathtt{H}$ ("can flow to")
  - extension to multi-level security by generalisation to lattice
- Analysis (can be) based on event traces in Evt^*
  - security assignment $\sigma : Evt \to \{\mathtt{L}, \mathtt{H}\}$
  - 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 $t_1|_E = t_2|_E$

Some Security Concepts

- Here: two security levels $\mathtt{L}$ (low/public) and $\mathtt{H}$ (high/confidential/secret/private)
  - partial order $\mathtt{L} \sqsubseteq \mathtt{H}$ ("can flow to")
  - extension to multi-level security by generalisation to lattice
- Analysis (can be) based on event traces in Evt^*
  - security assignment $\sigma : Evt \to \{\mathtt{L}, \mathtt{H}\}$
  - 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 $Evt = In \uplus Out$ and $T \subseteq Evt^*$. Security assignment σ ensures (event) non-interference if, for all $t_1, t_2 \in T$,

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

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!

Cryptographically-Masked Information Flow

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

Example (Password encryption)

- $In = \{pwd1_{\mathtt{H}}, pwd2_{\mathtt{H}}\}$, $Out = \{enc1_{\mathtt{L}}, enc2_{\mathtt{L}}\}$
- $t_1 = pwd1 \cdot enc1$, $t_2 = pwd2 \cdot enc2$
- $t_1|_{In \cap s^{-1}(\mathtt{L})} = \varepsilon = t_2|_{In \cap s^{-1}(\mathtt{L})}$, but $t_1|_{Out \cap s^{-1}(\mathtt{L})} = enc1 \neq enc2 = t_2|_{Out \cap s^{-1}(\mathtt{L})}$

$\Rightarrow$ Interference

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

Information Flow Security

Cryptographically-Masked Information Flow

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

Example (Password encryption)

- $\mathit{In} = \{\mathit{pwd1}_{\mathtt{H}}, \mathit{pwd2}_{\mathtt{H}}\}$, $\mathit{Out} = \{\mathit{enc1}_{\mathtt{L}}, \mathit{enc2}_{\mathtt{L}}\}$
- $t_1 = \mathit{pwd1} \cdot \mathit{enc1}$, $t_2 = \mathit{pwd2} \cdot \mathit{enc2}$
- $t_1|_{\mathit{In} \cap s^{-1}(\mathtt{L})} = \varepsilon = t_2|_{\mathit{In} \cap s^{-1}(\mathtt{L})}$, but $t_1|_{\mathit{Out} \cap s^{-1}(\mathtt{L})} = \mathit{enc1} \neq \mathit{enc2} = t_2|_{\mathit{Out} \cap s^{-1}(\mathtt{L})}$

$\Rightarrow$ 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

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

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)

```
m0 -[then low1 := encrypt(val, key)]-> m1;
m1 -[when high then low2 := encrypt(val, key)]-> m2;
m1 -[when not high then low2 := low1]-> m2;
```

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)

```
m0 -[then low1 := encrypt(val, key)]-> m1;
m1 -[when high then low2 := encrypt(val, key)]-> 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 if sets of possible results coincide

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

$\sim_{\mathrm{L}}$ is a low-equivalence relation on ciphertexts if $\forall v_1, v_2, k_1, k_2$:

1. safe usage: $\forall u_1 \in \mathtt{encrypt}(v_1, k_1). \exists u_2 \in \mathtt{encrypt}(v_2, k_2) : u_1 \sim_{\mathrm{L}} u_2$
2. prevent occlusion: $\exists u_1 \in \mathtt{encrypt}(v_1, k_1), u_2 \in \mathtt{encrypt}(v_2, k_2) : u_1 \not\sim_{\mathrm{L}} u_2$

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

$\sim_{\mathrm{L}}$ is a low-equivalence relation on ciphertexts if $\forall v_1, v_2, k_1, k_2$:

1. safe usage: $\forall u_1 \in \mathtt{encrypt}(v_1, k_1).\exists u_2 \in \mathtt{encrypt}(v_2, k_2) : u_1 \sim_{\mathrm{L}} u_2$
2. prevent occlusion: $\exists u_1 \in \mathtt{encrypt}(v_1, k_1), u_2 \in \mathtt{encrypt}(v_2, k_2) : u_1 \not\sim_{\mathrm{L}} u_2$

- Lifted  to low-equivalence relation $\sim_{\mathrm{L}}$ 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

Possibilistic Non-Interference and Safe Usage of Encryption

Example (Safe usage of encryption)

```
m0 -[then low := encrypt(high, key)]-> m1;
```

- Let $\sigma(\texttt{high}) = \texttt{H}$ and $\sigma(\texttt{key}) = \sigma(\texttt{low}) = \texttt{L}$
- Let environments η_1, η_2 with $\eta_1 \sim_{\texttt{L}} \eta_2$ such that
  1. $\eta_1(\texttt{high}) = v_1$, $\eta_1(\texttt{key}) = k$
  2. $\eta_2(\texttt{high}) = v_2$, $\eta_2(\texttt{key}) = k$
- Execution respectively yields
  1. $E'_1 = \{\eta_1[\texttt{low} \mapsto u_1] \mid u_1 \in \texttt{encrypt}(v_1, k)\}$
  2. $E'_2 = \{\eta_2[\texttt{low} \mapsto u_2] \mid u_2 \in \texttt{encrypt}(v_2, k)\}$
- Now $\forall u_1 \in \texttt{encrypt}(v_1, k_1).\exists u_2 \in \texttt{encrypt}(v_2, k_2) : u_1 \sim_{\texttt{L}} u_2$ implies that $\forall \eta'_1 \in E'_1.\exists \eta'_2 \in E'_2 : \eta'_1 \sim_{\texttt{L}} \eta'_2$

$\Rightarrow$ Possibilistic non-interference

Possibilistic Non-Interference and Occlusion

Example (Occlusion)

```
m0 -[then low1 := encrypt(val, key)]-> m1;
m1 -[when high then low2 := encrypt(val, key)]-> m2;
m1 -[when not high then low2 := low1]-> m2;
```

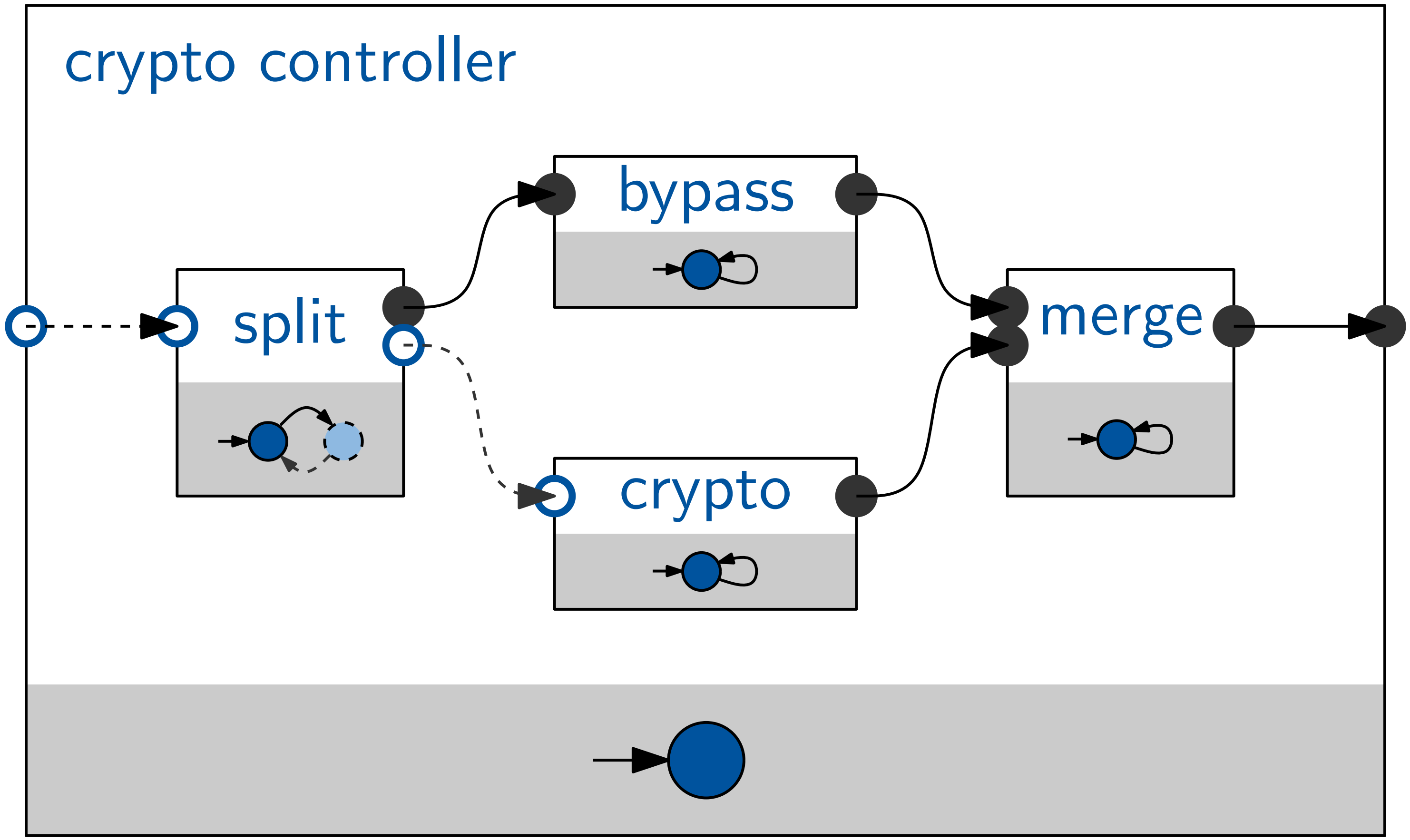
- Let $\sigma(\texttt{high}) = \sigma(\texttt{val}) = \texttt{H}$ and $\sigma(\texttt{key}) = \sigma(\texttt{low1}) = \sigma(\texttt{low2}) = \texttt{L}$
- Let environments η_1, η_2 with $\eta_1 \sim_{\texttt{L}} \eta_2$ such that
  1. $\eta_1(\texttt{high}) = \mathit{true}$, $\eta_1(\texttt{val}) = v_1$, $\eta_1(\texttt{key}) = k$
  2. $\eta_2(\texttt{high}) = \mathit{false}$, $\eta_2(\texttt{val}) = v_2$, $\eta_2(\texttt{key}) = k$
- Execution respectively yields
  1. $E'_1 = \{\eta_1[\texttt{low1} \mapsto u_1, \texttt{low2} \mapsto u_2] \mid u_1 \in \texttt{encrypt}(v_1, k), u_2 \in \texttt{encrypt}(v_2, k)\}$
  2. $E'_2 = \{\eta_2[\texttt{low1} \mapsto u, \texttt{low2} \mapsto u] \mid u \in \texttt{encrypt}(v_1, k)\}$
- Now $\exists u_1 \in \texttt{encrypt}(v_1, k), u_2 \in \texttt{encrypt}(v_2, k) : u_1 \not\sim_{\texttt{L}} u_2$ implies that $\exists \eta'_1 \in E'_1 : \eta'_1(\texttt{low1}) \not\sim_{\texttt{L}} \eta'_1(\texttt{low2})$
- On the other hand, $\forall \eta'_2 \in E'_2 : \eta'_2(\texttt{low1}) \sim_{\texttt{L}} \eta'_2(\texttt{low2})$
- Thus $\exists \eta'_1 \in E'_1 . \forall \eta'_2 \in E'_2 : \eta'_1 \not\sim_{\texttt{L}} \eta'_2$

$\Rightarrow$ Possibilistic interference

Content

The Type Checking Approach

MILS-AADL Specifications



Software Modeling and Verification Chair

RWTH AACHEN UNIVERSITY

The Type Checking Approach

- Introduce typing environment T
  - local variables and data ports $\longrightarrow$ security type τ (data type t + security level σ)
  - modes and event ports $\longrightarrow$ security level σ

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

The Type Checking Approach

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

$$\frac{T \vdash e_1 : \tau \quad T \vdash e_2 : \texttt{key L}}{T \vdash \texttt{encrypt}(e_1, e_2) : \texttt{enc}\ \tau\ \texttt{L}}$$

$$\frac{T \vdash e_1 : \texttt{enc}\ \tau\ \sigma \quad T \vdash e_2 : \texttt{key H}}{T \vdash \texttt{decrypt}(e_1, e_2) : \tau^{\sigma}}$$

The Type Checking Approach

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

$$\frac{T \vdash e_1 : \tau \quad T \vdash e_2 : \mathtt{key\ L}}{T \vdash \mathtt{encrypt}(e_1, e_2) : \mathtt{enc}\ \tau\ \mathtt{L}}$$

$$\frac{T \vdash e_1 : \mathtt{enc}\ \tau\ \sigma \quad T \vdash e_2 : \mathtt{key\ H}}{T \vdash \mathtt{decrypt}(e_1, e_2) : \tau^{\sigma}}$$

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

Content

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 $\sigma(\texttt{low}) = \texttt{L}$ 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 $\sigma(\texttt{low}) = \mathtt{L}$ 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:

$$\frac{T \vdash e_1 : \mathtt{enc}\ (\mathtt{int}\ \mathtt{H})\ \mathtt{L} \quad T \vdash e_2 : \mathtt{key}\ \ \mathtt{H}}{T \vdash \mathtt{decrypt}(e_1, e_2) : \mathtt{int}\ \mathtt{H}}$$

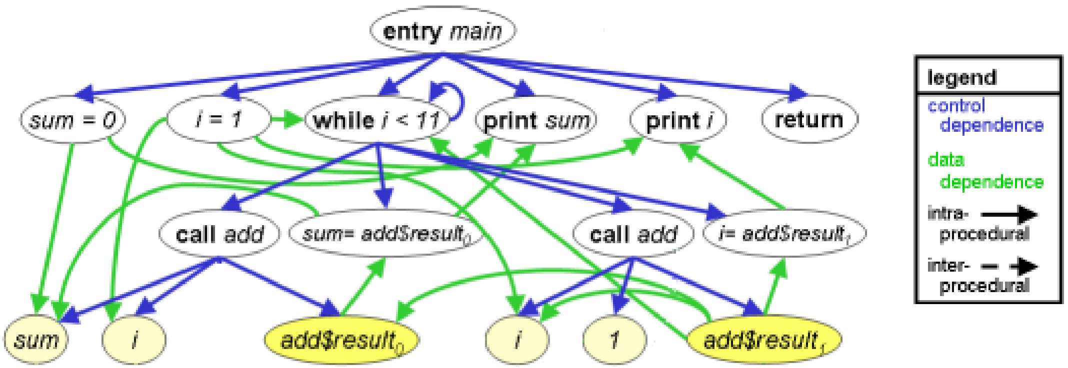
yields $\sigma(\mathtt{decrypt}(e_1, e_2)) = \mathtt{H}$ even if e_2 cannot be the matching private key

The Slicing Approach

Slicing

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

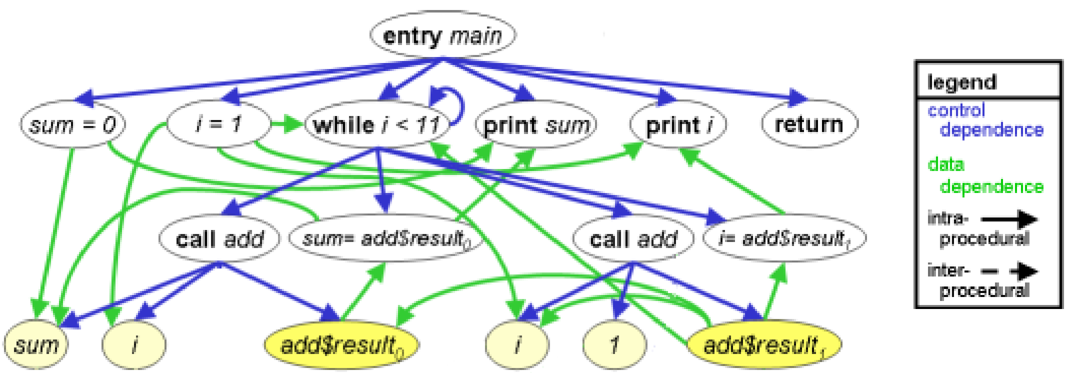
- interesting output values define slicing criterion
- backward analysis of information flow based on program dependence graph
- analysis inherently flow-sensitive!



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

Slicing AADL Specifications for Model Checking [NFM 2010]

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

repeat

until nothing changes;

Slicing AADL Specifications for Model Checking [NFM 2010]

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

repeat

for all $m \xrightarrow{e,g,f} m' \in \mathit{Trn}$ 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;

Slicing AADL Specifications for Model Checking [NFM 2010]

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

repeat

for all $m \xrightarrow{e,g,f} m' \in \mathit{Trn}$ 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

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

} Transitions from/to interesting modes

until nothing changes;

Slicing AADL Specifications for Model Checking [NFM 2010]

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

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**
$M := M \cup \{m\};$

} Transitions that affect interesting data elements or have interesting triggers

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

} Transitions from/to interesting modes

for all $a \rightsquigarrow d \in Flw$ with $d \in D$ **do**
$D := D \cup \{d' \in Dat \mid a \text{ reads } d'\};$
$M := M \cup \{m \in Mod \mid d := a \text{ active in } m\};$

} Data flows to interesting ports

until nothing changes;

Slicing AADL Specifications for Model Checking [NFM 2010]

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

repeat

for all $m \xrightarrow{e,g,f} m' \in \mathit{Trn}$ 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

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

} Transitions from/to interesting modes

for all $a \rightsquigarrow d \in \mathit{Flw}$ with $d \in D$ **do**
$D := D \cup \{d' \in \mathit{Dat} \mid a \text{ reads } d'\};$
$M := M \cup \{m \in \mathit{Mod} \mid d := a \text{ active in } m\};$

} Data flows to interesting ports

for all $e \rightsquigarrow e' \in \mathit{Con}$ with $e \in E$ or $e' \in E$ **do**
$E := E \cup \{e, e'\};$
$M := M \cup \{m \in \mathit{Mod} \mid e \rightsquigarrow e' \text{ active in } m\};$

} Connections involving interesting event ports

until nothing changes;

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
)
```

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`}

```
 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 sources and modes of flows with interesting targets

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 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
 )
 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

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

```
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

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

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 sources and modes of flows with interesting targets

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
)
```

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

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 sources and modes of flows with interesting targets

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
)
```

Thus: (low) `outframe` depends on (high) `inframe` $\Longrightarrow$ (classical) interference!

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

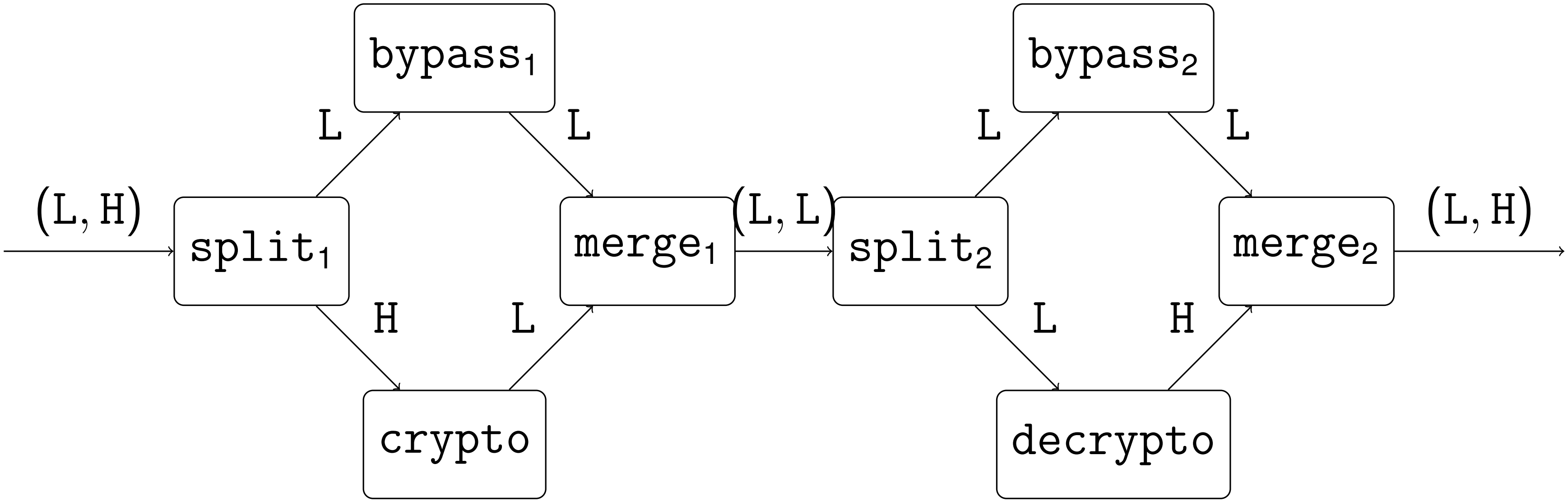
  $\Rightarrow$ static pool of keys with dynamic distribution

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

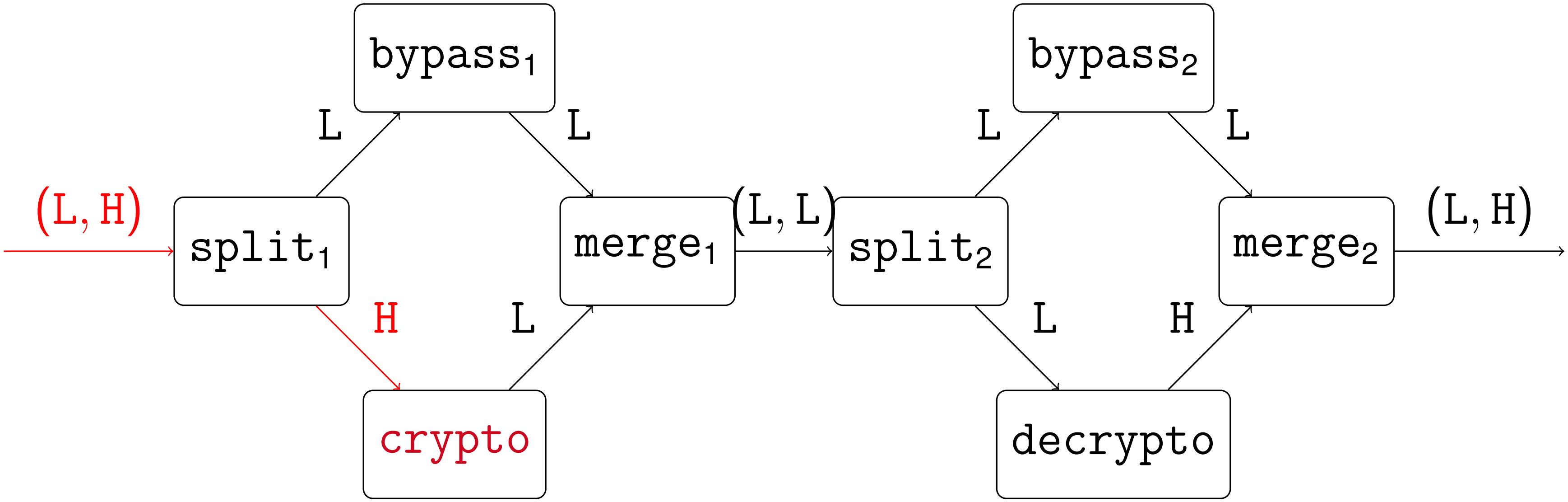
  ⇒ static pool of keys with dynamic distribution
- Analysis approach: conditional slicing w.r.t. knowledge of keys
  - attach security level to each data element (ports and subcomponents)
  - `encrypt(val,key)`:
    - maintain sets of data elements (D) and public keys (U) that *may* be used in first/as 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'

Example: Secure Communication



The Slicing Approach

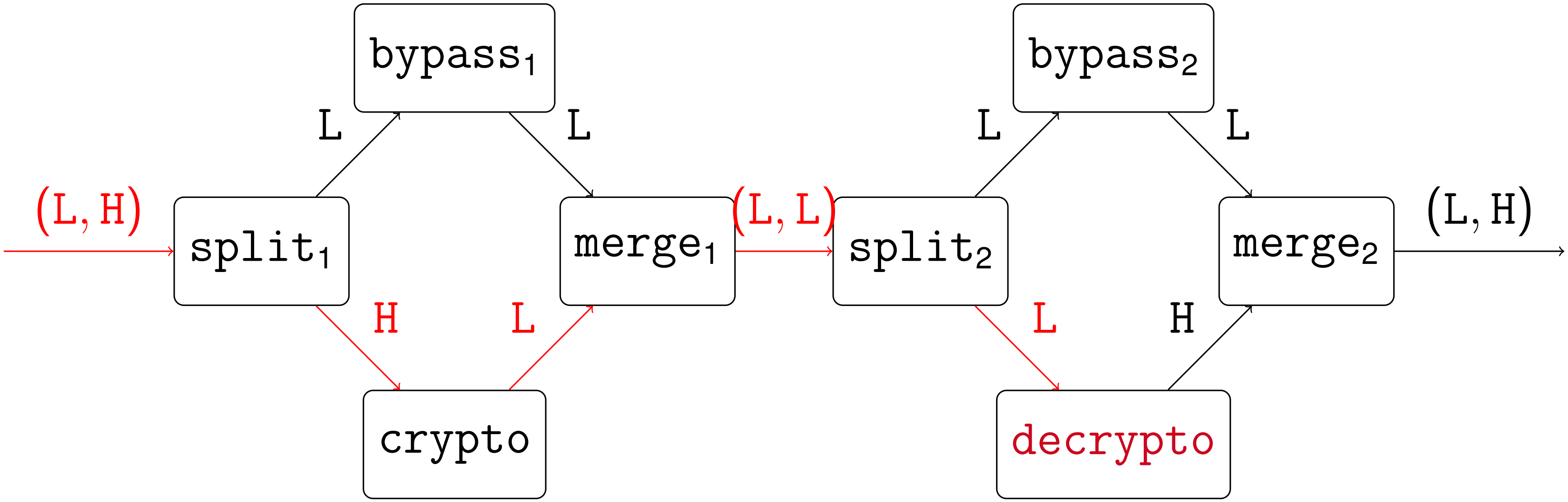
Example: Secure Communication



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

The Slicing Approach

Example: Secure Communication



1. `outpayload := encrypt(inpayload,k1)` with $\texttt{k1} = \texttt{pub(mykeys)}$
   - $D = \{\texttt{split}_1\texttt{.payload}, \texttt{split}_1\texttt{.frame}, \texttt{inframe}\}$
   - $U = \{\texttt{mykeys}\}$
2. `outpayload := decrypt(inpayload,k2)` with $\texttt{k2} = \texttt{priv(mykeys)}$
   - $P = \{\texttt{mykeys}\}$

$\Rightarrow P \cap U = \{\texttt{mykeys}\} \neq \emptyset$

$\Rightarrow D' = \{\texttt{split}_1\texttt{.payload}, \texttt{split}_1\texttt{.frame}, \texttt{inframe}\}$

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?