Analysing Cryptographically-Masked Information Flows in MILS-AADL Specifications

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

MILS Workshop @ embedded world Conference

Nuremberg, Germany; March 14, 2017

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

Content

Context: The D-MILS Project

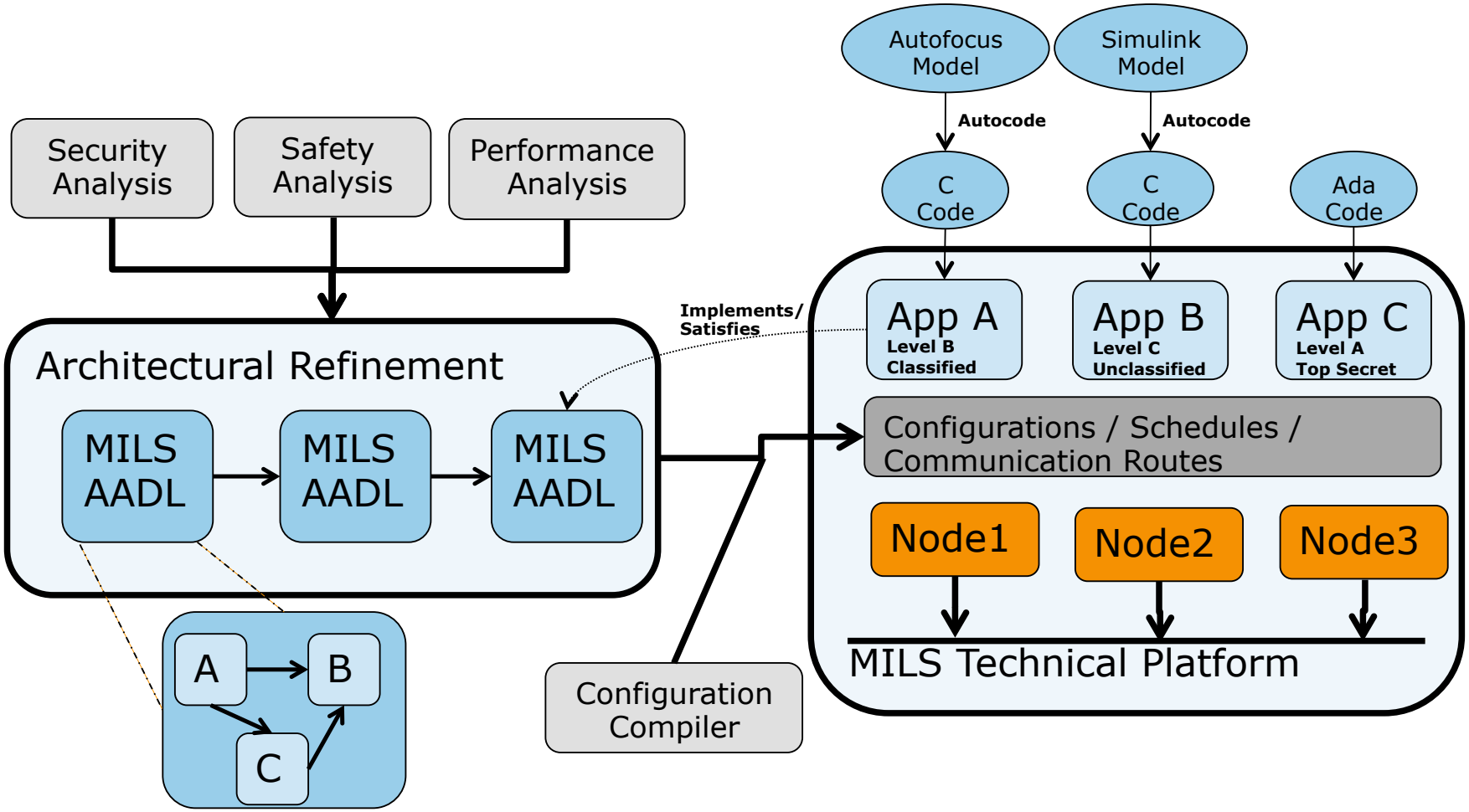
Information Flow Security

The Slicing Approach

Handling Cryptographic Operations

Context: The D-MILS Project

D-MILS Design Flow



Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

MILS Policy Architecture

The architecture expresses an interaction policy among a collection of components

Circles represent architectural components (subjects / objects)

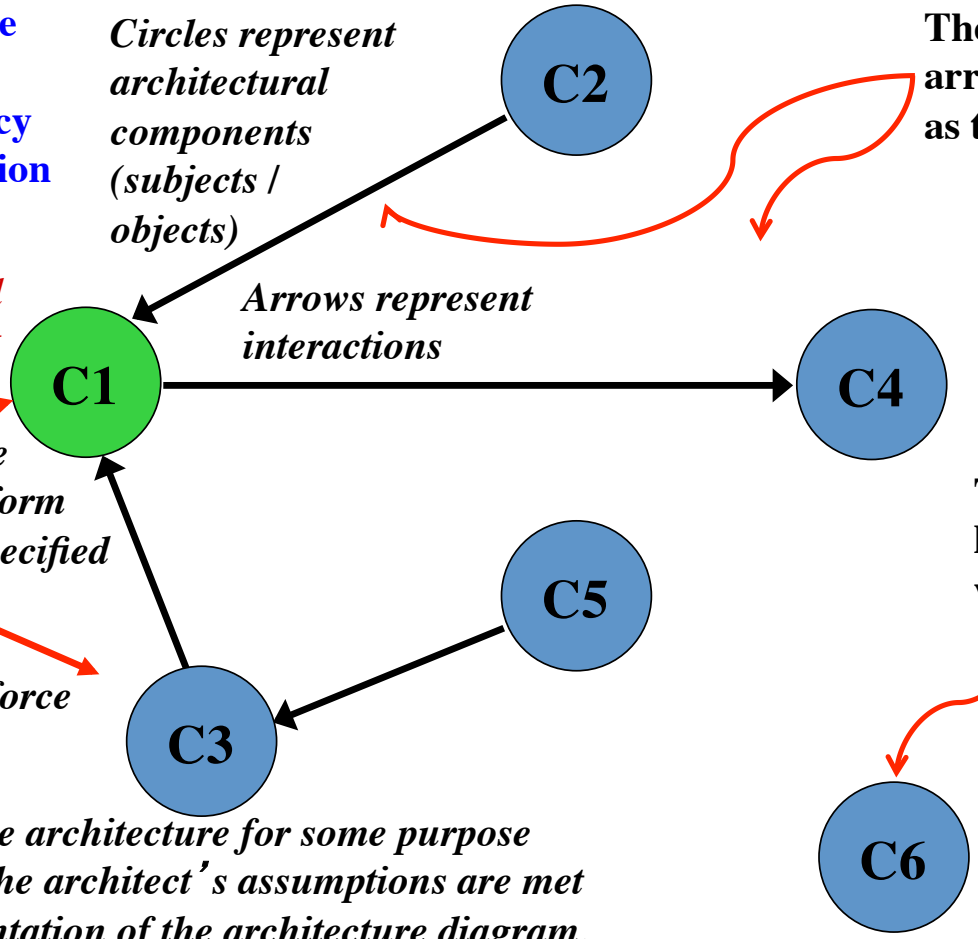
The *absence* of an arrow *is as significant* as the *presence* of one

Trusted Subject

Arrows represent interactions

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.

Content

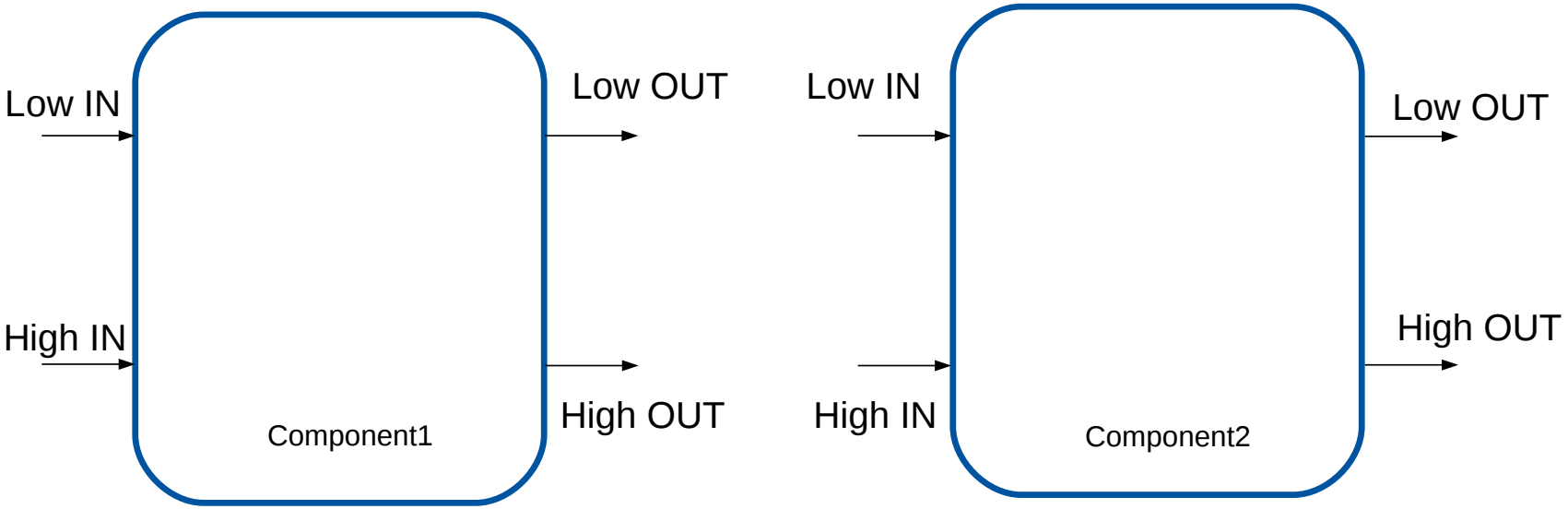
Context: The D-MILS Project

Information Flow Security

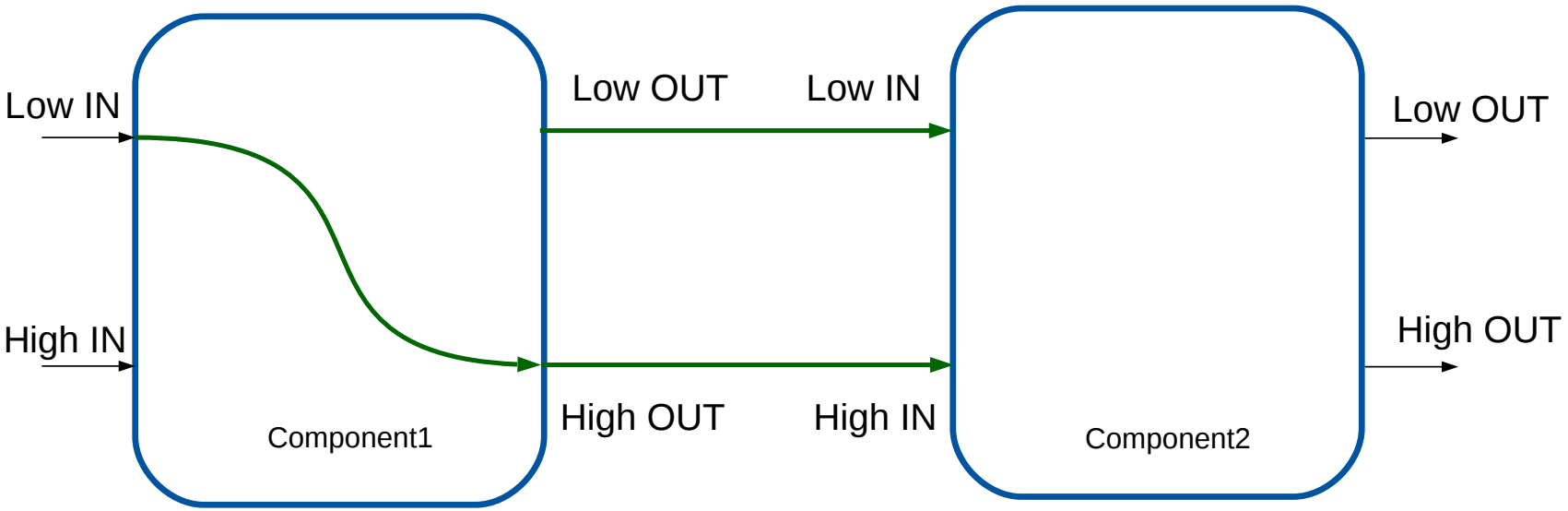
The Slicing Approach

Handling Cryptographic Operations

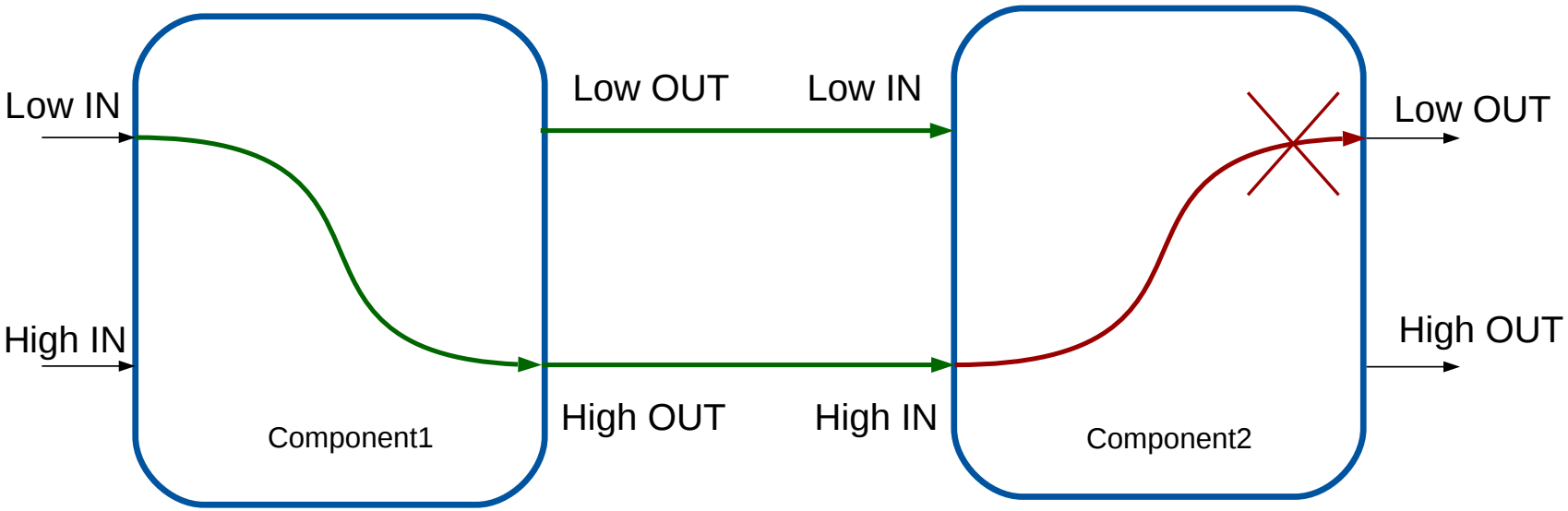
Information Flows



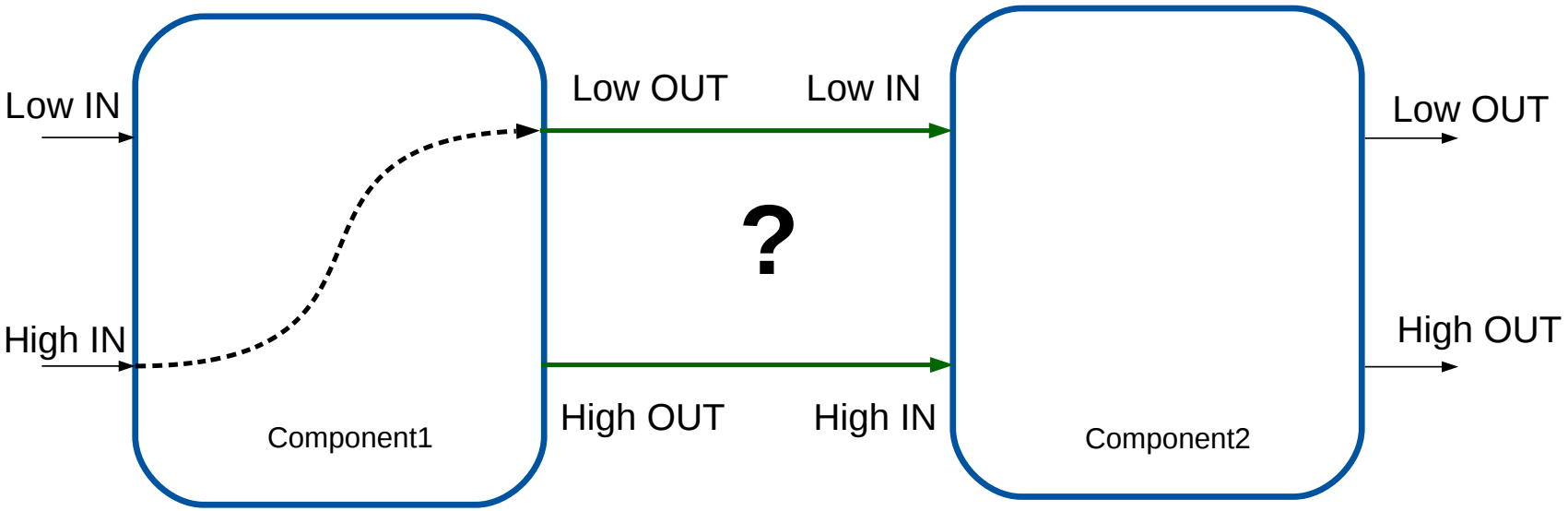
Legal Flows


Low IN
High IN
Component1
Low OUT
High OUT
Low IN
High IN
Component2
Low OUT
High OUT

Illegal Flows



Cryptographically-Masked Flows



Non-Interference

Security levels

- L (low/public)
- H (high/confidential/secret/private)
- Partial order $L \sqsubseteq H$ ("can flow to")
- Security assignment for attaching security levels to system actions/data/events/...
- Extension to multi-level security by generalisation to lattice

Non-Interference

Security levels

- L (low/public)
- H (high/confidential/secret/private)
- Partial order $L \sqsubseteq H$ ("can flow to")
- Security assignment for attaching security levels to system actions/data/events/...
- Extension to multi-level security by generalisation to lattice

Non-interference [Goguen/Meseguer 1982]

Low outputs may not change when high inputs are varied, that is, behaviour seen by "low" observer must be unaffected by changes in "high" behaviour

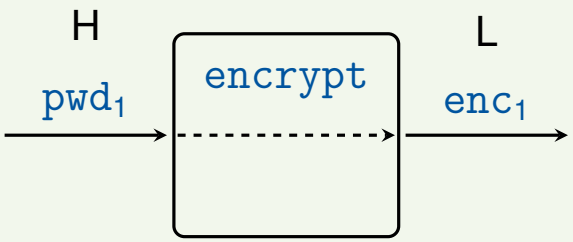
Cryptographically-Masked Information Flow

- **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!

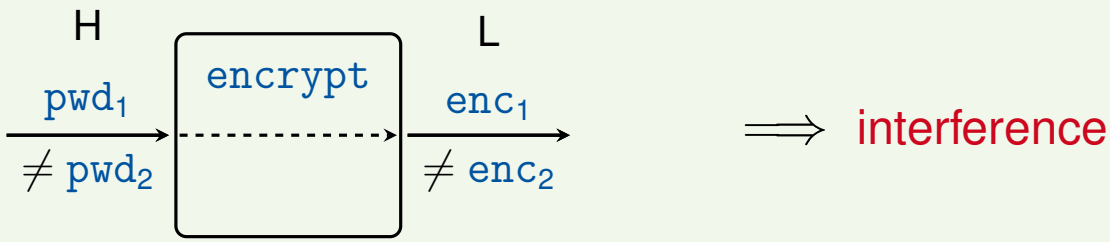
Example (Password encryption)



Cryptographically-Masked Information Flow

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

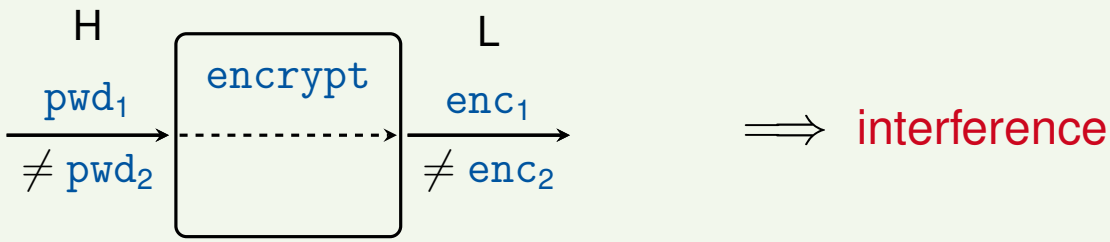
Example (Password encryption)



Cryptographically-Masked Information Flow

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

Example (Password encryption)



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 on two low-equivalent inputs, the resulting outputs 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)

Adapting Non-Interference

- Non-interference: if a program is run on two low-equivalent inputs, the resulting outputs 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)

```
low1 := encrypt(val1, key1);
if high then low2 := encrypt(val2, key2)
else low2 := low1 fi
```

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

Adapting Non-Interference

- Non-interference: if a program is run on two low-equivalent inputs, the resulting outputs 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)

```
low1 := encrypt(val1, key1);
if high then low2 := encrypt(val2, key2)
else low2 := low1 fi
```

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

Possibilistic Non-Interference

- 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

Possibilistic Non-Interference

- 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

Possibilistic non-interference [McCullough 1988]

If a program is run on two low-equivalent inputs, there exists a possibility that each output produced from the first input is low-equivalent to some that can be produced from the second input

Possibilistic (Non-)Interference

Example

1. Safe usage of encryption:

   ```
   low := encrypt(high, key)
   ```

   Modifying `high` does not change possible ciphertexts
   $\Longrightarrow$ possibilistic non-interference

Possibilistic (Non-)Interference

Example

1. Safe usage of encryption:

```
low := encrypt(high, key)
```

   Modifying `high` does not change possible ciphertexts
   $\Longrightarrow$ possibilistic non-interference

2. Occlusion:

```
low1 := encrypt(val1, key1);
if high then low2 := encrypt(val2, key2)
else low2 := low1 fi
```

   - `high` set: all combinations of ciphertexts in `low1`/`low2`
   - `high` unset: same ciphertext in `low1`/`low2`

   $\Longrightarrow$ possibilistic interference

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

Content

Context: The D-MILS Project

Information Flow Security

The Slicing Approach

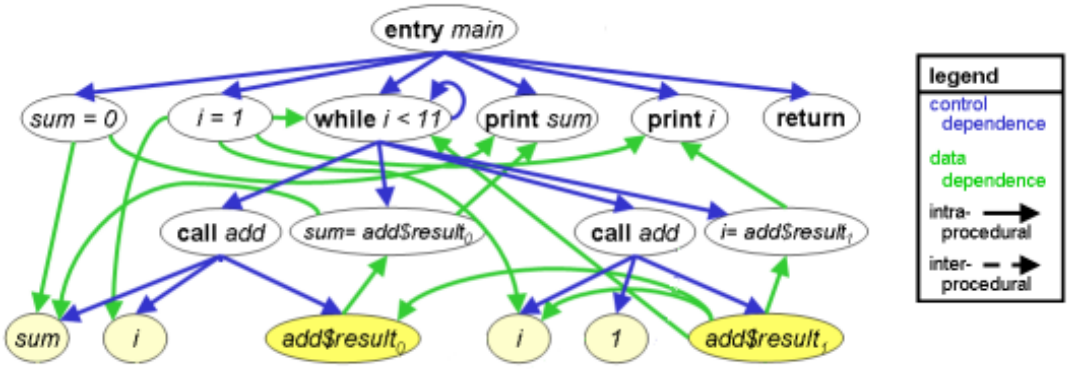
Handling Cryptographic Operations

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



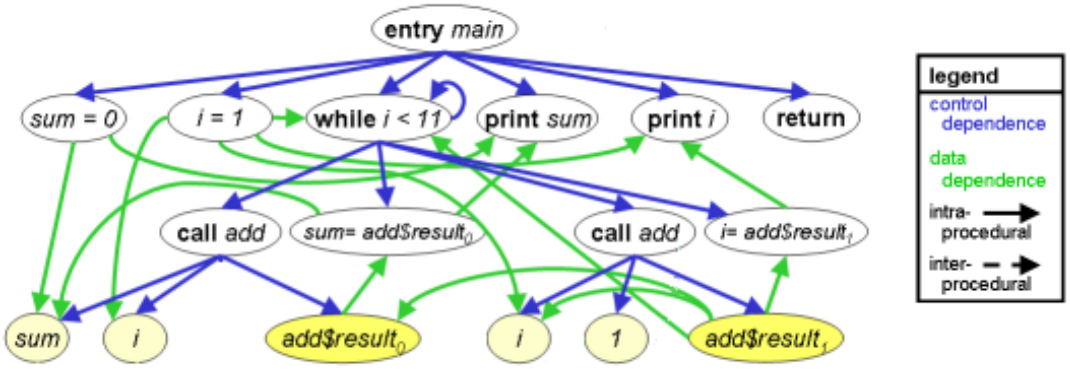
Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

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

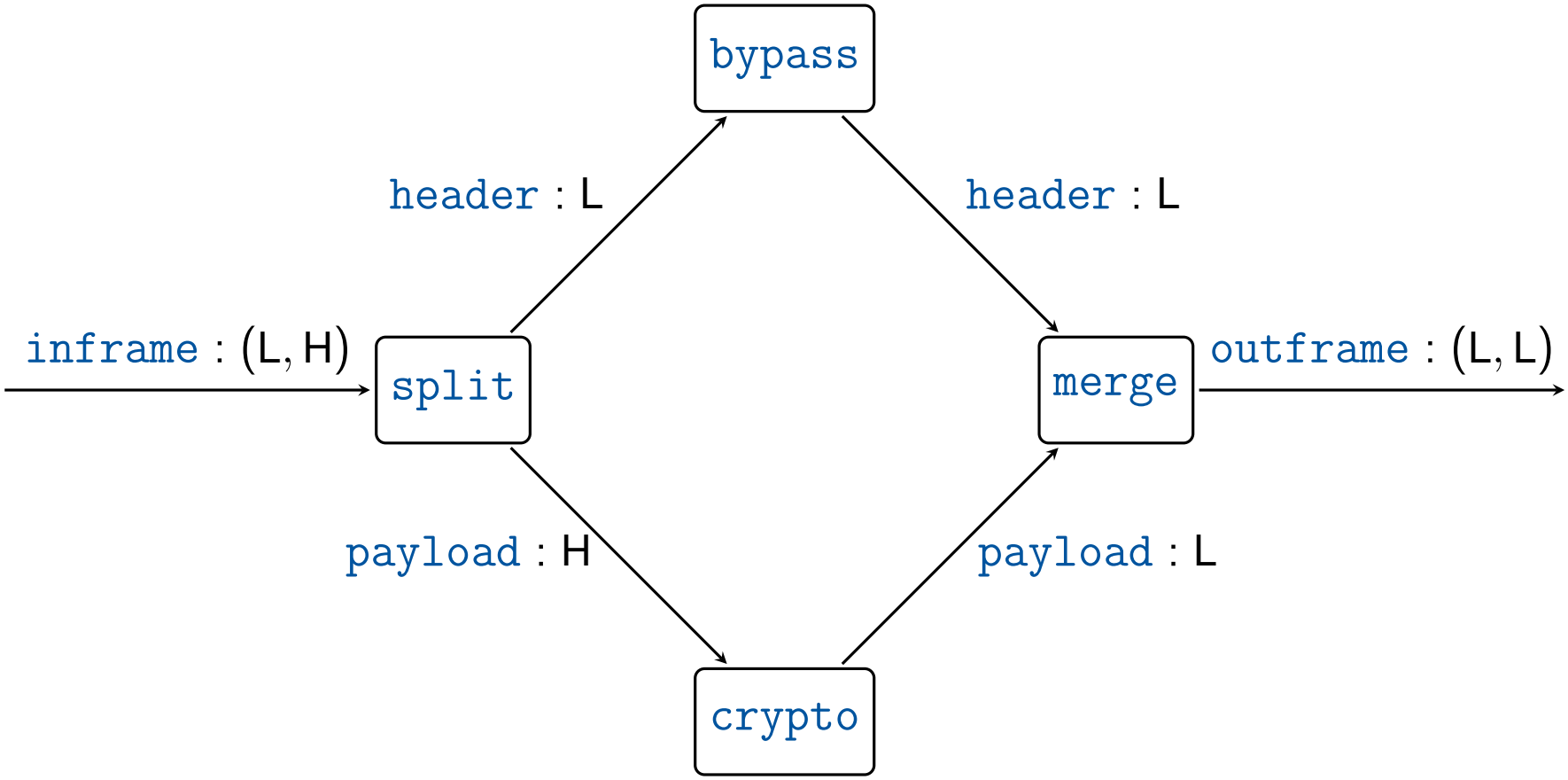
Software security [Snelting et al.]

Relation to non-interference:
no high variable in backward slice of any low output

$\Longrightarrow$

system is non-interfering

Example: A Crypto Controller



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
)
```

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
)
```

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

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

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

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

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

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

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

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

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

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

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
)
```

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

Content

Context: The D-MILS Project

Information Flow Security

The Slicing Approach

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

$\Rightarrow$ 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: 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 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

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: `encrypt(val,key)`

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

Handling Cryptographic Operations

Conditional Slicing

Analysis approach

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

Encryption: `encrypt(val,key)`

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

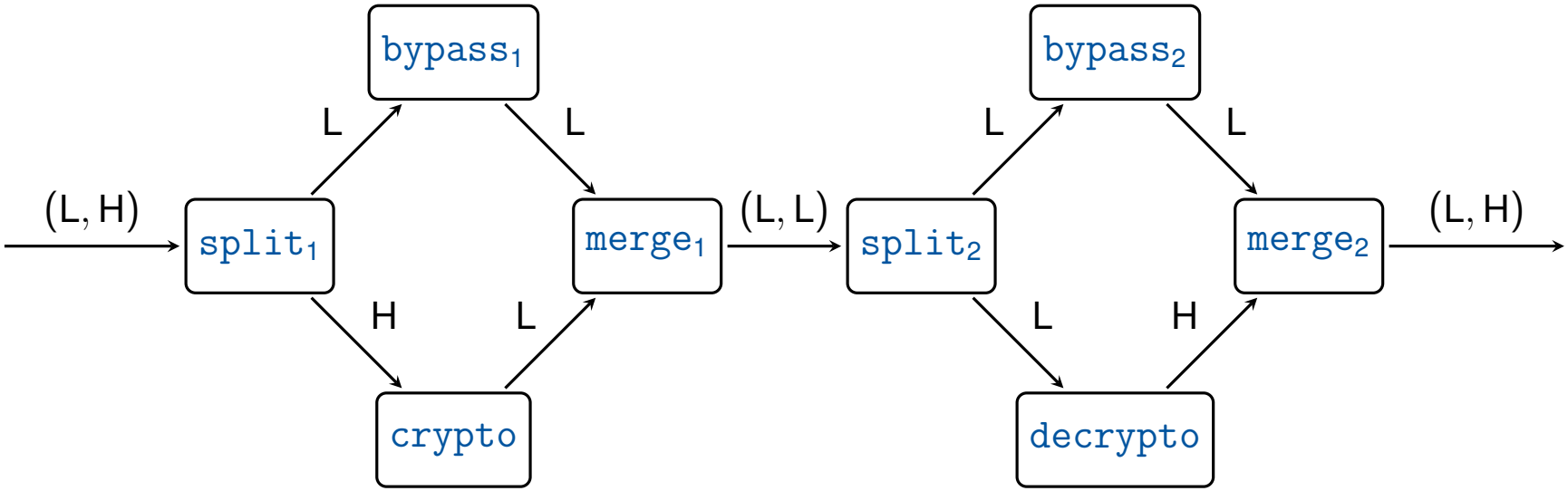
Decryption: `decrypt(val,key)`

- maintain sets of (C, U)-pairs and private keys (P) that *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'

Conditional Slicing Algorithm

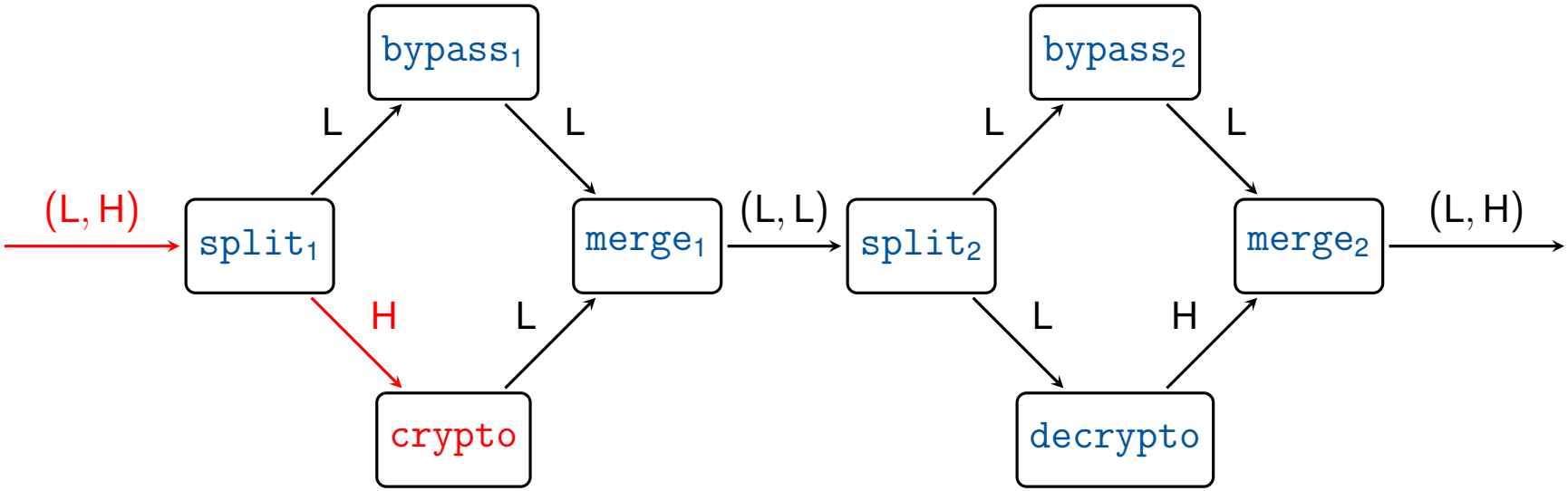
$D := S$; $CU := \emptyset$; $E := \emptyset$; $M := \emptyset$; /* Input: slicing criterion $S \subseteq Dat$ */
repeat
  for all $m \xrightarrow{e,g,f} m' \in Trn$ with $\exists d \in D : f$ updates d or $\exists d \in D : d$ inactive in m but active in m' or $e \in E$ **do**
    $M := M \cup \{m\}$;
  for all $m \xrightarrow{e,g,f} m' \in Trn$ with $m \in M$ or $m' \in M$ **do**
    $D := D \cup \{d \in Dat \mid g \text{ reads } d\}$; $E := E \cup \{e\}$; $M := M \cup \{m\}$;
    for all $d' := a$ in f with $d' \in D$ **do**
      if $a = \texttt{encrypt}(d, k)$ **then**
        $(C, CU', E', M') := CondSlice(\{d\})$; $(U, CU'', E'', M'') := CondSlice(\{k\})$;
        $CU := CU \cup \{(C, U)\}$; $E := E \cup E' \cup E''$; $M := M \cup M' \cup M''$;
      else if $a = \texttt{decrypt}(d, k)$ **then**
        $(D', CU', E', M') := CondSlice(\{d\})$; $(P, CU'', E'', M'') := CondSlice(\{k\})$;
        $D := D \cup \{C \mid (C, U) \in CU', U \cap P \neq \emptyset\}$; $E := E \cup E' \cup E''$; $M := M \cup M' \cup M''$;
      else
        $D := D \cup \{d \mid a \text{ reads } d\}$;
  for all $a \rightsquigarrow d' \in Flw$ with $d \in D$ **do**
    $D := D \cup \{d \in Dat \mid a \text{ reads } d\}$; $M := M \cup \{m \in Mod \mid a \rightsquigarrow d' \text{ active in } m\}$;
  for all $e \rightsquigarrow e' \in Con$ with $e \in E$ or $e' \in E$ **do**
    $E := E \cup \{e, e'\}$; $M := M \cup \{m \in Mod \mid e \rightsquigarrow e' \text{ active in } m\}$;
until nothing changes;
return (D, CU, E, M)

Example: Secure Communication



Handling Cryptographic Operations

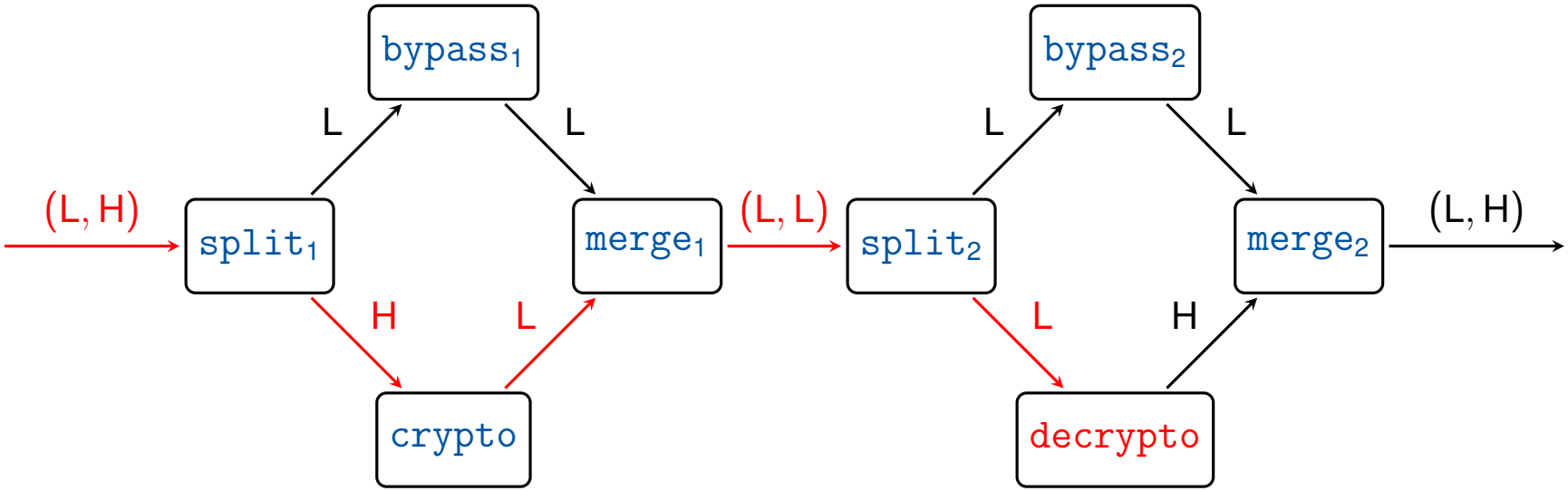
Example: Secure Communication



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

Handling Cryptographic Operations

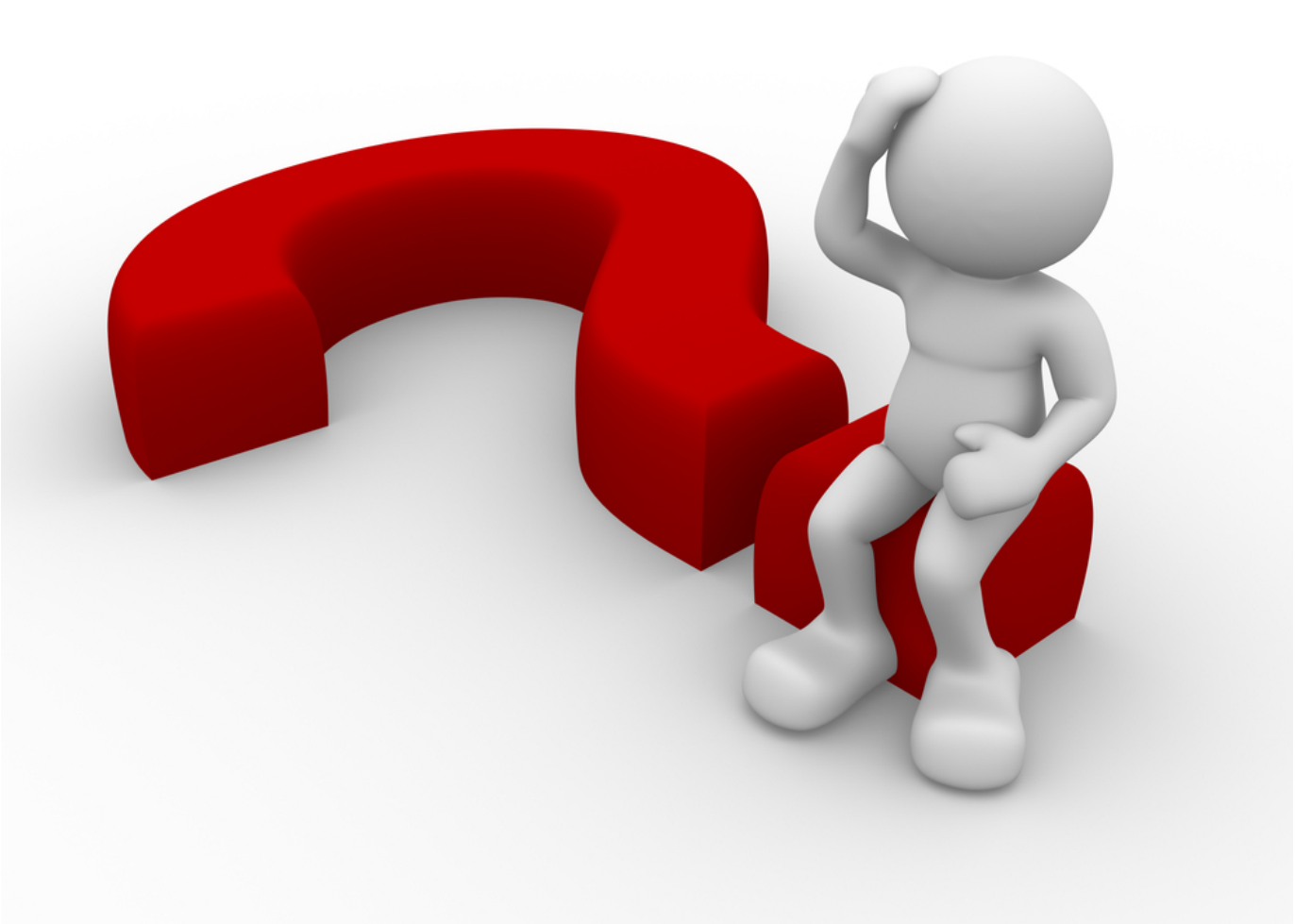
Example: Secure Communication



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

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

   $\Rightarrow C = \{\texttt{split}_1\texttt{.payload}, \texttt{split}_1\texttt{.frame}, \texttt{inframe}\}$ added to D

The End

Questions?