Security Type Checking for MILS-AADL Specifications*

Kevin van der Pol
Software Modelling and Verification Group
RWTH Aachen University
52056 Aachen, Germany
kvdpol@cs.rwth-aachen.de

Thomas Noll
Software Modelling and Verification Group
RWTH Aachen University
52056 Aachen, Germany
noll@cs.rwth-aachen.de

ABSTRACT
Information flow policies are widely used for specifying confidentiality and integrity requirements of security-critical systems. In contrast to access control policies and security protocols, they impose global constraints on the information flow and thus provide end-to-end security guarantees. The information flow policy that is usually adopted is non-interference. It postulates that confidential data must not affect the publicly visible behavior of a system. However, this requirement is usually broken in the presence of cryptographic operations.

In this paper, we provide an extended definition of non-interference for systems that are specified in a MILS variant of the Architecture Analysis and Design Language (AADL). More concretely, we propose a type system for MILS-AADL component definitions that distinguishes between breaking non-interference because of legitimate use of sufficiently strong encryption and breaking non-interference due to an unintended information leak. To this aim, it tracks both intra- and inter-component information flow and considers both data- and event-flow security.

Keywords
Security, MILS components, information flow, type system

1. INTRODUCTION
Modern critical systems bear great responsibilities and face escalating challenges that require the assurance of security, safety, and other dependability attributes. With respect to the verification of security properties, most existing techniques concentrate on *access control policies* and *security protocols* that are essential for ensuring data confidentiality and integrity, but do not provide end-to-end security guarantees. Access control policies do not track information flow through the entire system and do not cope with implicit information flow. Similarly, cryptographic components are used for secure communication and authentication but do not guarantee any global security properties.

Information flow policies [3, 4, 8] are natural for specifying end-to-end confidentiality and integrity requirements because they impose global constraints on the information flow. For example, an access control policy can require that only users with the appropriate rights can read a file while an information flow policy would require that only users with the appropriate security level can get any information about the content of a file, even indirectly. Verification of such properties has to meet the following challenges:

- Information flow properties are more complex than safety and liveness properties because they are defined in terms of sets of possible system traces.
- The analysis has to take into account malicious agents that try to corrupt the system.
- To scale, the verification has to be modular, i.e., the implementation of each (benevolent) component should be separately analyzable.

The information flow policy that is usually adopted is *non-interference*. This notion has first been defined in [8] in order to capture the presence of illegal flows from a group of users to another. It requires that confidential data must not affect the publicly visible behavior of the system. In other terms, if the security levels are classified as being either low or high, then a low-level user must not be able to detect the values of the high-level data by observing the low-level data behavior.

This traditional notion of non-interference can be very non-intuitive, and it is almost impossible to find real systems without interfering data. For example, a classical password verification communication is interferent, as the value of the public data (server's response) depends on the secret input (password). In general, the requirement that public outputs are unchanged as secret inputs are varied is usually broken in the presence of cryptographic operations. Thus, the challenge is to distinguish between breaking non-interference because of legitimate use of sufficiently strong encryption (so-called cryptographically-masked information flows) and breaking non-interference due to an unintended information leak.

*The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP 7/2007–2013] under grant agreement no. 318772 – *Distributed MILS for Dependable Information and Communication Infrastructures (D-MILS)*

Case	Grammar
Type	τ ::= `int` | `bool` | `key` | `enc` τ | $(\tau \ldots, \tau)$
Expression	$e ::= n \mid x \mid e \oplus e \mid (e, \ldots, e) \mid e[n]$
System	S ::= `system` $s(S^* \; P^* \; C^* \; V^* \; M^* \; T^*)$
Port	$P ::= p :$ (`in` | `out`)(`event` | `data` τ e)
Connection	$C ::= ([s.]p, [s.]p)$
Variable	$V ::= x : \tau \; e$
Mode	$M ::= m :$ [`initial`] `mode`
Transition	$T ::= m$ `-[` $[p]$ [`when` e] [`then` $x := e$] `]->` m'

Table 1: MILS-AADL syntax

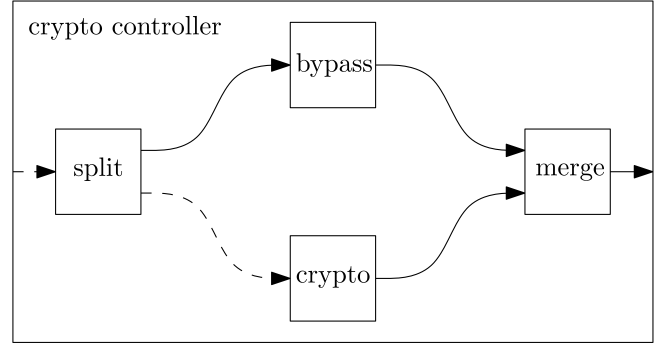


Figure 1: *The architecture of the crypto controller system of Example 1. The dashed lines indicate that confidential data travels along these connections. This figure is only intended to give an overview of the crypto controller architecture.*

In this paper, we provide an extended definition of non-interference for systems that are specified in a MILS variant of the Architecture Analysis and Design Language (AADL; [16]). This language has been developed within the D-MILS project [1] and is entitled MILS-AADL [5]. We propose a type system for MILS-AADL system definitions that prevents dangerous system behaviour. To this aim, it tracks both intra- and inter-component information flow and considers both data- and event-flow security.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of the MILS-AADL language. Section 3 describes the relevant security concepts. Section 4 introduces the type system. Section 5 sketches a correctness proof of the claim that well-typed specifications are non-interfering. Finally, Section 6 draws some conclusions and points to ongoing and future work.

2. THE MILS-AADL LANGUAGE

The specification language MILS-AADL [5] has been developed within the D-MILS project and is intended to serve as the user-facing representation for model-based design of D-MILS systems. Essential features are component definitions in terms of interfaces and implementations, their architecture and interaction through data and event ports, their internal behavior, and security-related mechanisms such as encryption and authentication.

2.1 Syntax

We analyze models described in a simplified version of MILS-AADL, a modeling language based on the AADL language. The full MILS-AADL language is described in detail in [5], our simplified version can be found in Table 1. MILS-AADL combines an architectural and behavioral description of a system s, as follows: one describes a hierarchy of systems (merging 'component types' and 'component implementation' in the full MILS-AADL language), each possibly containing subsystems, input ports, output ports and connections between ports of different systems. The highest or outermost system in this hierarchy is called the root system. This describes the architecture. We distinguish event ports and data ports. Event ports can trigger changes in behavior. Data ports are used to communicate data values to or from the environment. This is the behavior, described as an automaton with 'modes' and 'transitions' between them. Transitions are labeled with an *event* given by an event port p (input/output events are consumed/produced when the transition is taken, respectively), a *guard* expression (the transition is enabled only when the guard evaluates to true) and a list of *effects* $x_1 := e_1; \ldots; x_n := e_n$ (expressions are evaluated in the source mode and assigned simultaneously to these variables). In this paper, we assume there is only one effect per transition. The event, guard and effects are all optional. If the event is omitted, no port's event will be consumed or produced. If the guard is omitted, it equals true. If the effect is omitted, the system's variables remain unchanged.

Specifically with respect to security, MILS-AADL provides *security primitives* in its expression language. Most importantly for this paper, there is encrypt(m, k), taking a message m of some type τ and a public key k : `key` and producing a ciphertext of type `enc` τ. The original message can be decrypted from the ciphertext using the decrypt function: decrypt$(c, k') : \tau$ takes a ciphertext c : `enc` τ and a private key k' : `key` to reproduce the message. If k' is the matching private key to the public key k used for encryption, this message is the original message m. Otherwise, decryption fails and the statement containing the decryption expression deadlocks.

Example 1. The running example of this paper is taken from [12]. A cryptographic controller is placed between a secure computer and an untrusted network, encrypting all data going from the secure computer to the untrusted network. However, only the payloads of the messages are considered confidential, not their headers. We want to ensure the confidentiality of the payload. A visual representation of this system's architecture is given in Figure 1.

```
system cryptocontroller(
 inframe: in data (int,int) (0,0)
 outframe: out data (int,enc int) (0,encrypt(0,k0))
 m0: initial mode
 system split(
  frame: in data (int,int) (0,0)
  header: out data int 0
  payload: out data int 0
  m0: initial mode
  m0 -[then header:=frame[0];
           payload:=frame[1]]-> m0
 )
```

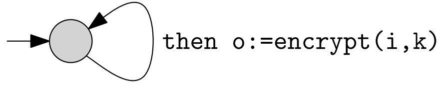


Figure 2: The crypto system.

```
 system bypass(
  iheader: in data int 0
  oheader: out data int 0
  m0: initial mode
  m0 -[then outheader:=inheader]-> m0
 )
 system crypto(
  inpayload: in data int 0
  outpayload: out data enc int encrypt(0,k0)
  k: key k0
  m0: initial mode
  m0 -[then outpayload:=encrypt(inpayload,k)]-> m0
 )
 system merge(
  header: in data int 0
  payload: in data enc int encrypt(0,k0)
  frame: out data (int,enc int) (0,encrypt(0,k0))
  m0: initial mode
  m0 -[then frame:=(header, payload)]-> m0
 )
 connection (inframe, split.frame)
 connection (split.header, bypass.inheader)
 connection (split.payload, crypto.inpayload)
 connection (bypass.outheader, merge.header)
 connection (crypto.outpayload, merge.payload)
 connection (merge.frame, outframe)
)
```

This illustrates how systems can be combined to larger systems. For the remainder of this paper, we consider the `crypto` system in isolation, as it contains the relevant security primitives we want to focus on, with `inpayload` abbreviated to `i` and `outpayload` to `o`.

We visually represent modes with circles and mode transitions with arrows between them. Modes and transitions with a high confidentiality, which will be explained later, are drawn with dashed lines. The `crypto` system's behavior is illustrated in Figure 2. *(End of example 1.)*

Other security primitives included in full MILS-AADL are signing and hashing, which are not discussed in this paper. For signing, this is because integrity is fully dual to confidentiality. The analysis laid out in this paper to secure systems against data leaks, can be straightforwardly applied to secure systems against breaches of integrity. For hashing, this is because hashing provides only quantitative security and we are interested in qualitative security. An attacker can guess the contents of a hashed value and test if this guess is correct. The assumption that this guess will almost certainly be incorrect is what makes the hash functions secure in a quantitative setting. In our qualitative setting, however, we have to consider *any* information leak impermissible. Volpano [17] notes that "[secure type rules for hashes] are certainly not sound with respect to non-interference".

2.2 Semantics

The semantics of full MILS-AADL are formally defined in [6]. We present the semantics directly as a *labeled transition system* (LTS). The states are given by the modes of all (sub)systems and the values of the data ports and variables. The transitions are given by the mode transitions and changes in input data ports under the control of the environment. The label is the event if it is present, an internal action otherwise. We denote internal actions by omitting the label. In the case of input data ports, it is the port and the new value. Not all mode transitions lead to a corresponding transition in the LTS. A mode transition is enabled in the LTS if and only if the guard evaluates to true in the source mode, and

1. there is no event,
2. the event is an input event port and the system is the root system, or
3. the event is an input event port and is connected to another system's output event port, taking a transition on this event *simultaneously*,

and vice versa for output ports. Input data ports are controlled by the environment and can change their value at any time. Output data ports change their value in an effect. Input event ports can provide an event to trigger a transition with that port as event. Output event ports emit such an event when a transition is taken with that port as event.

For data port connections, values at the source end (an output port, or an input port going to a subsystem's input port) travel instantaneously to the corresponding target end. This is included in the transition that changes the data value at the source end. Similarly for event port connections: the event travels along the connection instantaneously.

Example 2. The semantics of the `crypto` system of Example 1 is an LTS, where the state space is the cartesian product of the mode and the possible values for each input and output data port and each variable. In the example, this is the cartesian product of the singleton set of modes $\{\texttt{m0}\}$, the integers (for `i`), the ciphertexts with integer contents (for `o`), and the keys (for `k`), which we denote by the set $\mathbb{K}$:

$$S = \{m_0\} \times \mathbb{Z} \times \{\texttt{encrypt}(z, k) \mid z \in \mathbb{Z}, k \in \mathbb{K}\} \times \mathbb{K}$$

The initial mode is given by the mode marked 'initial' and the default values in the declarations of the data ports and variables:

$$s_0 = (m_0, 0, \texttt{encrypt}(0, k_0), k_0)$$

Finally, the transition relation $\rightarrow$ is the union of two cases: first, input data ports can change their value at any time. All other values remain equal. So, for any m, i, i', o, k, we have a transition

$$(m, i, o, k) \xrightarrow{\texttt{i},i'} (m, i', o, k)$$

Second, the mode transition from m_0 to itself can be taken spontaneously (as the triggering event is omitted), updating the output port `o`. For any i, o, k, we have:

$$(m_0, i, o, k) \rightarrow (m_0, i, \texttt{encrypt}(i, k), k)$$

(End of example 2.)

3. SECURITY

This section introduces the relevant security concepts. First, we make precise what we mean by confidentiality levels. This allows us to further clarify non-interference and introduce the concept of possibilistic non-interference. Then we explain what attackers can observe, by defining the low equivalence relation. Finally, we explain the problems of non-interference in a context with non-determinism and pose restrictions on our language to resolve these problems.

3.1 Confidentiality levels

A *confidentiality level* describes what data is confidential and what is public.

W.l.o.g., we assume that there are two confidentiality levels, `H` (high) and `L` (low). Intuitively, the high confidentiality level means that information is to be kept secret and not made visible to the outside world. No such restriction is placed on data with a low confidentiality level. We occasionally say some data is 'secret' to mean it has a high confidentiality level. The data marked `L` may also be considered `H` (but certainly not the other way round!), denoted `H` $\sqsubseteq$ `L`. We assume the relation $\sqsubseteq$ is a preorder. This relation will be used in Section 4.2 to define subtyping.

For integrity, there is a dual *integrity level*. Where confidentiality restricts the flow of high-confidentiality variables to untrusted output channels, integrity restricts the flow of untrusted input channels to high-integrity variables. Since integrity is fully dual to confidentiality, we omit it for brevity.

3.2 Possibilistic non-interference

Standard non-interference states that low confidentiality outputs may not change when high confidentiality inputs are changed. In a system such as the crypto controller example of Section 1, this is violated. However, this system could be considered safe because encryption masks the information from any attackers. Thus, standard non-interference rejects arguably secure and intended uses of encryption.

A different variant of non-interference aims to repair this defect. In *possibilistic non-interference* [11], we look at the possible values after encryption instead of the actual value. We assume that the result of encryption is possibly *any* value in the ciphertext domain. Varying the contents now does not change the possible outcomes of encryption: any ciphertext is still a possible public output. Encryption is an instance of *declassification* [15], where an expression which depends on secret values is not itself secret.

Care must be taken, however, that not all ciphertexts are the same. Simply considering all ciphertexts as equivalent observations to an attacker opens the door for a problem known as *occlusion* [14]. Occlusion is when a declassification mechanism is abused to 'launder' secret data not intended for declassification. We consider a class of occlusion problems called *implicit data flow* [13], where secret data becomes observable through its influence on the control flow. This is in contrast to *explicit data flow*, where secret data is directly assigned to public variables.

We illustrate implicit data flow with the following specification (adapted from [2]):

```
system implicitdataflow(
  s: bool //secret
  v: int  //secret
  k: key  //secret
  o1: out enc int //public
  o2: out enc int //public
  m0: initial mode
  m1: mode
  m2: mode
  m0 -[then o1 := encrypt(v,k)]-> m1
  m1 -[when s then o2 := encrypt(v,k)]-> m2
  m1 -[when not s then o2 := o1]-> m2
)
```

In this specification, inequality of `o1` and `o2` reflects the secret `s`. Even though in both occurrences of `encrypt(v,k)` all ciphertexts are possible, the actual values could also differ, which is not possible in the assignment `o2 := o1`.

To combat implicit flows, we must be more careful what ciphertexts we declare to be indistinguishable to an attacker. We re-use the result from [2], where they discuss this specific issue. They assume an indistinguishability equivalence on ciphertexts, $\doteq$, such that for any v, k, v', k':

$$\forall u \in \texttt{encrypt}(v,k) \ . \ \exists u' \in \texttt{encrypt}(v',k') \ . \ u \doteq u', \text{ and}$$

$$\exists u \in \texttt{encrypt}(v,k), u' \in \texttt{encrypt}(v',k') \ . \ u \not\doteq u'.$$

We refer to that paper for the following results on this indistinguishability relation on ciphertexts:

- The first condition allows for safe usages in the sense of possibilistic encryption. It excludes the existence of a ciphertext $u \in \texttt{encrypt}(v,k)$ that is distinguishable from any $u' \in \texttt{encrypt}(v',k')$.
- The second condition prevents implicit flow. It states that not all ciphertexts are indistinguishable.
- Both conditions are realistic for encryption schemes with the computational security properties Indistinguishability under Chosen Plaintext attack (IND-CPA) and Integrity of Plaintexts (INT-PTXT) (by a previous result from Laud [10]).

3.3 Low equivalence

We use the indistinguishability equivalence on ciphertexts to define what attackers can observe under possibilistic non-interference. The intuition is that attackers can see the value of low variables, but cannot distinguish between high variables. With possibilistic non-interference, we assume that attackers can observe that ciphertexts change when their contents change, but this still does not give them any information on what the contents are: some ciphertexts 'look the same' to the attacker. We formalize this by the *low equivalence relation* $\sim$, defined in Table 2.

Two values are low equivalent iff. under the given confidentiality level, the two values are indistinguishable to an attacker.

Case	Low equivalence	
Integers	$\overline{n \sim_{\texttt{int L}} n}$	$\overline{n \sim_{\texttt{int H}} n'}$
Booleans	$\overline{b \sim_{\texttt{bool L}} b}$	$\overline{b \sim_{\texttt{bool H}} b'}$
Tuple	$\dfrac{v_1 \sim_{\tau_1} v_1' \quad \dots \quad v_n \sim_{\tau_n} v_n'}{(v_1, \dots, v_n) \sim_{(\tau_1,\dots,\tau_n)} (v_1', \dots, v_n')}$	
Public keys	$\overline{k_{\text{public}} \sim_{\texttt{key L}} k_{\text{public}}}$	
Private keys	$\overline{k_{\text{private}} \sim_{\texttt{key H}} k'_{\text{private}}}$	
Ciphertext	$\dfrac{\begin{array}{ll} k_1 \sim_{\texttt{key H}} k_2 & \\ \exists v_1, k_1 . v_1 = \text{decrypt}(u_1, k_1) & v_1 \sim_{\tau} v_2 \\ \exists v_2, k_2 . v_2 = \text{decrypt}(u_2, k_2) & u_1 \doteq u_2 \end{array}}{u_1 \sim_{\texttt{enc } \tau \texttt{ L}} u_2}$	

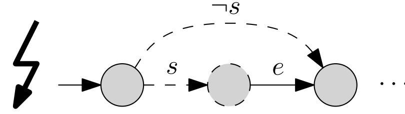
Table 2: Low equivalence

3.4 Non-determinism

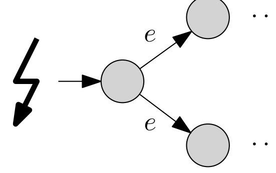
It has been shown that in a non-deterministic setting, non-interference is *not compositional* [11]. In other words, the composition of two non-interfering systems, is not necessarily non-interfering. The counterexample to non-interference compositionality given in that paper, are two communicating systems which are secure individually because they can non-deterministically choose between two alternatives, hiding whether a change in output is due to this choice or a secret input. When composed, however, this non-deterministic choice becomes observable and non-interference is broken. Put differently, the problem here is that the *strategy* to resolve the non-determinism has to remain secret. This problem is related to the so-called *refinement paradox* [9]: non-interference is not preserved under refinement.

Compositionality of non-interference is a useful property to have, as our model is compositional. Preservation of non-interference under refinement is useful for software engineering. We avoid both of these problems by excluding non-determinism emanating from high-confidentiality data, except for the specific case of non-deterministic encryption (discussed in Section 3.2). Since our expressions do not contain non-deterministic choice, the unwanted non-determinism we wish to exclude, can only come from interleaving or multiple transitions being enabled.

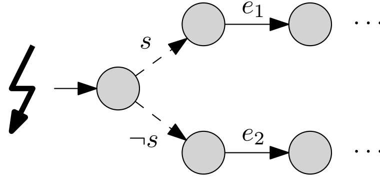
This non-determinism is only a problem for modes that deal with confidential data, which we will mark as high confidentiality modes, and we require that the transitions from high confidentiality modes to other modes are not observable. This is illustrated in Figure 3a. High and low confidentiality modes will be explained later in Section 4. Systems must be deterministic in the following sense: we require that the path of low confidentiality modes, i.e. observable parts of the path, does not depend on non-determinism from interleaving or multiple transitions being enabled. Specifically, at most one transition may be enabled by an event, and if a transition leaves a low mode using any high confidentiality information, the next low confidentiality mode on the path is fully determined by the initial low mode. Thus, a model in which the event and transition guards overlap is not allowed (cf. Figure 3b).



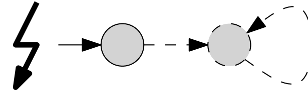
(a) Transitions from a confidential mode to another mode, may not be observable. In this example, the value of the boolean secret s can be inferred from observing transition e. This is verified by type checking (cf. Section 4.4).



(b) Non-determinism from overlapping events or guards is not allowed.



(c) The next low confidentiality mode on a path must be determined by the initial low mode. In this example, the boolean secret s can be learned from observing transition e_1 or e_2.



(d) Zenoness restriction. Performing high confidentiality actions indefinitely is not allowed. In this example, entering the confidential self-loop is (eventually) observable to an attacker, even without explicit observable transitions.

Figure 3: *Restrictions on the models. We verify through type checking that the first restriction holds. The other restrictions are assumed to hold, but we do not verify this.*

A model in which two transitions with different secret events (or non-overlapping guards dependent on secret variables) are enabled, leading to different low confidentiality modes, is not allowed, either (cf. Figure 3c).

We will not verify these non-determinism restrictions ourselves. Automatic verification of these restrictions is possible, but outside of the scope of this paper. Rather, we assume the model obeys these restrictions and the user has independently verified this.

3.5 Restrictions

Similarly, non-interference can be broken by observing whether or not a program terminates. In our setting, models run indefinitely except when there is a deadlock or when decryption fails because the decryption key does not match the encryption key. We therefore additionally require that path segments using high information either always or never

Case	Grammar
Level	$\sigma ::= \texttt{H} \mid \texttt{L}$
Basic type	$t ::= \texttt{int} \mid \texttt{bool} \mid \texttt{enc}\ \tau$
Security type	$\tau ::= t\ \sigma \mid \texttt{key}\ \sigma \mid (\tau, \ldots, \tau)$
Expression	$e ::= n \mid x \mid e \oplus e \mid (e, \ldots, e) \mid e[n]$
System	$S ::= \texttt{system}\ s(S^*\ P^*\ C^*\ V^*\ M^*\ T^*)$
Port	$P ::= p : (\texttt{in} \mid \texttt{out})(\texttt{event}\ \sigma \mid \texttt{data}\ \tau\ e)$
Connection	$C ::= ([s.]p, [s.]p)$
Variable	$V ::= x : \tau\ e$
Mode	$M ::= m : [\texttt{initial}]\ \texttt{mode}\ \sigma$
Transition	$T ::= m\ \texttt{-[}\ [p]\ [\texttt{when}\ e]\ [\texttt{then}\ x := e]\ \texttt{]->}\ m'$

(Symbols with changed definitions appear in *italics*)

Table 3: MILS-AADL syntax with security types

reach some transition with an observable action.

Moreover, we employ a condition related to the Zeno phenomenon, where it must never be possible to do high confidentiality actions indefinitely, leading to no observations for an attacker. This restriction is taken from [7], who note that "no realistic model of information flow could accept [high users] making infinitely many actions without letting [low users] observe this". This is illustrated in Figure 3d.

Since termination checking is no small problem, we will not attempt to verify these restrictions ourselves. As for the non-determinism requirements, we assume the user has independently verified that the model obeys these restrictions.

4. THE TYPE SYSTEM

To describe what data is confidential and what is public, we enhance the type system described in Section 2 by attaching a *security type* to the relevant syntactic constructs. We then type check the annotated model to verify that it is non-interfering.

4.1 Syntax

The user annotates each mode, port and variable with a confidentiality level. For port and variable values, this confidentiality level can be lowered by encrypting. The new type syntax is described in Table 3. These annotations yield a function T mapping data ports and variables to their declared type, and mapping modes and event ports to their declared confidentiality level.

For keys, we fix the confidentiality level as follows: private keys are always `H`, public keys are `L`. A generalized confidentiality level for keys is possible, but does not add any interesting cases.

We use the function lvl to give the confidentiality level of a type. Formally, lvl is defined as follows:

$$lvl(t\ \sigma) := \sigma$$
$$lvl(\texttt{key}\ \sigma) := \sigma$$
$$lvl((\tau_1, \ldots, \tau_n)) := lvl(\tau_1) \sqcup \cdots \sqcup lvl(\tau_n)$$

We lift the definition of lvl to expressions in the obvious way. Moreover, for a mode m, let $lvl(m)$ be its declared confidentiality level, and similarly for an event port p.

Case	Type rule
Integer	$\dfrac{}{T \vdash n : \texttt{int L}}$
Boolean	$\dfrac{}{T \vdash b : \texttt{bool L}}$
Variable	$\dfrac{T(x) = \tau}{T \vdash x : \tau}$
Tuple	$\dfrac{T \vdash e_1 : \tau_1 \quad \ldots \quad T \vdash e_n : \tau_n}{T \vdash (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n)}$
Projection	$\dfrac{1 \le i \le n \quad T \vdash (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n)}{T \vdash (e_1, \ldots, e_n)[i] : \tau_i}$
Operator	$\dfrac{T \vdash e_1 : t_1\ \sigma_1 \quad T \vdash e_2 : t_2\ \sigma_2 \quad \oplus : t_1 \times t_2 \to t}{T \vdash e_1 \oplus e_2 : t\ (\sigma_1 \sqcup \sigma_2)}$
Encryption	$\dfrac{T \vdash e_1 : \tau \quad T \vdash e_2 : \texttt{key L}}{T \vdash \text{encrypt}(e_1, e_2) : \texttt{enc}\ \tau\ \texttt{L}}$
Decryption	$\dfrac{T \vdash e_1 : \texttt{enc}\ \tau\ \sigma \quad T \vdash e_2 : \texttt{key H}}{T \vdash \text{decrypt}(e_1, e_2) : \tau^{\sigma}}$

Table 4: Type rules for expressions

Example 3. We annotate the `crypto` system of Example 1 with confidentiality levels. The incoming payload `i` is confidential, i.e. of type `int H`. The outgoing payload is the incoming payload encrypted with `k`. The ciphertext's contents are of type `int H`, but the ciphertext itself has a low confidentiality. So, the outgoing payload `o` is of type `enc int H L`. The encryption key is public, i.e. `key L`. The mode is also annotated. We will explore confidentiality levels of modes later. The annotated example is as follows:

```
system crypto(
 i: in data int H 0
 o: out data enc int H L encrypt(0,k0)
 k: key L k0
 m0: initial mode L
 m0 -[then o := encrypt(i,k)]-> m0
)
```

The corresponding function T is as follows:

$$T(\texttt{i}) = \texttt{int H}$$
$$T(\texttt{o}) = \texttt{enc int H L}$$
$$T(\texttt{k}) = \texttt{key L}$$
$$T(\texttt{m0}) = \texttt{L}$$

(End of example 3.)

For a system annotated with confidentiality levels, we verify non-interference by type checking.

4.2 Expressions

We first consider type checking for expressions. The type rules for expressions can be found in Table 4. For obvious reasons, the type rules are considered in the context of T. Integers and booleans are of a low confidentiality by default. They can be considered secret through subtyping, as

Case	Subtype rule
Integer	$\dfrac{\sigma \sqsubseteq \sigma'}{\texttt{int}\ \sigma <: \texttt{int}\ \sigma'}$
Boolean	$\dfrac{\sigma \sqsubseteq \sigma'}{\texttt{bool}\ \sigma <: \texttt{bool}\ \sigma'}$
Keys	$\dfrac{}{\texttt{key}\ \sigma <: \texttt{key}\ \sigma}$
Tuple	$\dfrac{\tau_1 <: \tau_1' \quad \dots \quad \tau_n <: \tau_n'}{(\tau_1, \dots, \tau_n) <: (\tau_1', \dots, \tau_n')}$
Encryption	$\dfrac{\tau <: \tau' \quad \sigma \sqsubseteq \sigma'}{\texttt{enc}\ \tau\ \sigma <: \texttt{enc}\ \tau'\ \sigma'}$

Table 5: Subtyping

discussed below. Tuples and projection are standard. For operators, we consider total binary operators where the basic types are standard and the confidentiality level is the highest of the operands' confidentiality levels. Encryption lowers the confidentiality level to L, i.e., ciphertexts can be made public. Decryption expressions are of the type of the contents *tainted* with the confidentiality level of the ciphertext, where tainting a type τ with a security level σ, denoted τ^σ, is defined as:

$$
\begin{aligned}
(t\ \sigma)^{\sigma'} &:= t\ (\sigma \sqcup \sigma') \\
(\tau_1, \dots, \tau_n)^\sigma &:= (\tau_1^\sigma, \dots, \tau_n^\sigma) \\
(\texttt{key L})^{\texttt{L}} &:= \texttt{key L} \\
(\texttt{key H})^{\sigma} &:= \texttt{key H}
\end{aligned}
$$

Note that tainting keys with higher levels of confidentiality is not allowed. This is due to fixing the confidentiality levels of private and public keys: public keys cannot become private through tainting.

We denote that a type τ is a *subtype* of type τ' as $\tau <: \tau'$. The subtyping rules are fairly standard and can be found in Table 5. Subtyping allows us to succinctly describe that high confidentiality data may not be assigned to low confidentiality variables, i.e., restrict *explicit data flows.*

Example 4. Consider the `crypto` system of Example 3. The expression `encrypt(i,k)` is of type `enc int H L`, shown with the following derivation:

$$
\dfrac{\dfrac{T(\texttt{i}) = \texttt{int H}}{T \vdash \texttt{i} : \texttt{int H}} \quad \dfrac{T(\texttt{k}) = \texttt{key L}}{T \vdash \texttt{k} : \texttt{key L}}}{T \vdash \texttt{encrypt(i,k)} : \texttt{enc int H L}}
$$

(End of example 4.)

4.3 Connections

We need to restrict physical connections so that confidential ports are not physically connected to a public port. This is an instance of restricting explicit data flow. The corresponding type rule infers a confidentiality level for the connection only if it is properly connected. The corresponding type rule is:

$$
\dfrac{T \vdash s_1.p_1 : \tau_1 \quad T \vdash s_2.p_2 : \tau_2 \quad \tau_1 <: \tau_2}{T \vdash (s_1.p_1, s_2.p_2) : \tau_1} \ \text{(data connection)}
$$

For event ports, the type rule is similar:

$$
\dfrac{T \vdash s_1.p_1 : \sigma_1 \quad T \vdash s_2.p_2 : \sigma_2 \quad \sigma_1 \sqsubseteq \sigma_2}{T \vdash (s_1.p_1, s_2.p_2) : \sigma_1} \ \text{(event connection)}
$$

4.4 Modes and transitions

We now turn to type rules for modes and transitions. The aim is to reject systems with unwanted implicit data flows (cf. Section 3.2). We therefore attach a security level to the modes and transitions. For modes, we require that the user supplies this information. For transitions, we infer the confidentiality level from the source mode and the ports and variables it references.

Modes

The user declares modes as high or low confidentiality. Even though we pose no direct requirements on this declaration, the type rule for transitions indirectly restricts what modes may be declared to be of high or low confidentiality (cf. the remark at the end of this section). Intuitively, modes should be declared as high confidentiality if an attacker can gain knowledge about secret data from observing that the control flow is in this mode. For example, when a transition to mode m is triggered by a secret event, m is typically high.

Effects

We require that effects are properly typed. The effect is an assignment $x := e$, and we require that the type of x is a subtype of that of e. This enforces traditional type safety, e.g. one may not assign an integer value to a boolean variable. In our setting, by extending the type system with confidentiality levels, it also enforces explicit data flow security, e.g. one may not assign a confidential value to a public variable.

Example 5. For the `crypto` system of Example 3, we have shown in Example 4 that the expression `encrypt(i,k)` is of type `enc int H L`, which we temporarily abbreviate as τ in the derivation. We can use this to derive the type of the effect `o:=encrypt(i,k)`:

$$
\dfrac{\dfrac{T(\texttt{o}) = \tau}{T \vdash \texttt{o} : \tau} \quad \dfrac{\textit{(Example 4)}}{T \vdash \texttt{encrypt(i,k)} : \tau} \quad \dfrac{\dfrac{\texttt{H} \sqsubseteq \texttt{H}}{\texttt{int H} <: \texttt{int H}} \quad \texttt{L} \sqsubseteq \texttt{L}}{\tau <: \tau}}{T \vdash \texttt{o:=encrypt(i,k)} : \tau}
$$

(End of example 5.)

Transitions

Finally, we infer a confidentiality level of the transition itself which reflects the confidentiality level of its triggering event and guard expression. We require that transitions going from a high mode to any other mode do not assign any value to a low confidentiality variable or emit a low confidentiality event. The intuition is that observable behavior only occurs in the transitions from a low confidentiality mode to any other mode. First taking a transition from a high to a low confidentiality mode is allowed, keeping in mind the requirements of Section 3.5. This is somewhat similar to exiting a scope in imperative programming languages.

Moreover, we disallow assigning any value to a low confidentiality variable in a transition with a high confidentiality, as this makes the occurrence of the secret event or the value of

Case	Type rule
Transition	$\dfrac{\begin{array}{lll} T \vdash p : \tau_p & & \\ T \vdash g : \tau_g & \tau_x <: \tau_e & lvl(\tau_x) \sqsubseteq \sigma \\ T \vdash x : \tau_x & lvl(m) \sqsubseteq lvl(\tau_p) & \sigma \sqsubseteq lvl(\tau_p) \\ T \vdash e : \tau_e & lvl(m) \sqsubseteq lvl(\tau_x) & \sigma \sqsubseteq lvl(\tau_g) \end{array}}{T \vdash m\texttt{-[}p \texttt{ when } g \texttt{ then } x := e\texttt{]->}m' : \sigma}$

Table 6: Type rule for transitions

the confidential guard expression observable. This enforces implicit data flow security.

This is expressed in the type rule in Table 6.

In conclusion, we pose the following requirements on transitions:

1. The effect must be properly typed, i.e., the type of x is a subtype of that of e: $\tau_x <: \tau_e$,
2. a transition from a high mode to another mode may not be triggered by a low confidentiality event, i.e., the source mode has a higher confidential level than the event: $lvl(m) \sqsubseteq lvl(\tau_p)$,
3. a transition from a high mode to another mode may not assign any value to a low confidentiality variable, i.e., the source mode has a higher confidentiality level than the effect's variable: $lvl(m) \sqsubseteq lvl(\tau_x)$,
4. a high transition may not assign any value to a low confidentiality variable, i.e., the variable x has a higher confidential level than the transition: $lvl(\tau_x) \sqsubseteq \sigma$,
5. a transition is high if it is triggered by a high confidentiality event, i.e., the transition has a higher confidential level than the event: $\sigma \sqsubseteq lvl(\tau_p)$, and
6. a transition is high if its guard expression has a high confidentiality, i.e., the transition has a higher confidential level than the guard: $\sigma \sqsubseteq lvl(\tau_g)$.

Example 6. Finally, we show that the transition in Example 3 is typeable. This is shown by a derivation tree using the transition type rule. Because this tree would be rather large, we show the requirements individually:

- In Example 5 we have shown that the effect is properly typed.
- In the absence of a trigger event p, the requirements with $lvl(\tau_p)$ trivially hold: the requirement $lvl(m) \sqsubseteq lvl(\tau_p)$ prohibits breaching confidential control flow information by emitting a public event; the requirement $\sigma \sqsubseteq lvl(\tau_p)$ prohibits breaching a secret event's confidentiality.
- The confidentiality level of the source mode `m0` and the effect's variable `o` are both declared to be `L`, so $lvl(m) \sqsubseteq lvl(\tau_x)$ holds.
- As the confidentiality level of the effect's variable `o` is `L`, the requirement $lvl(\tau_x) \sqsubseteq \sigma$ implies that σ must also be `L`.
- There is an implicit guard `true`. Its type is `bool L`, so the requirement $\sigma \sqsubseteq lvl(\tau_g)$ holds for any σ.

This is consistent with a confidentiality level of $\sigma = \texttt{L}$ for the transition. This means the transition is typeable.

The system `crypto` does not contain any connections or subsystems, so they are trivially correctly typed. Had the system contained subsystems, we would recursively type check these.

In conclusion, since we have shown that all connections, subsystems and transitions are typeable, we conclude that the system `crypto` is non-interfering.

(End of example 6.)

Remark

Remember that we do not impose any restrictions on the confidentiality of modes directly. The user is free to declare any mode as high or low confidentiality, as long as the restrictions in Sections 3.4 and 3.5 are met and all transitions are typeable. Intuitively, all modes are of low confidentiality except when they deal with confidential information. If one were to make too many modes of a low confidentiality, the restrictions of Section 3.4 will reject any branching on confidential data. On the other hand, if one were to make too many modes of a high confidentiality, transitions which update public data will not be typeable. When the system is typeable, the confidential data cannot spill over into public outputs, which is exactly what we aim for.

5. CORRECTNESS

This section shows the soundness of our approach. Since this paper presents work in progress, we formulate our lemmas and theorems as conjectures, accompanied by a proof sketch.

5.1 Overview

We want to show that systems for which all subsystems, connections and transitions are typeable, are non-interfering. A system is non-interfering iff. a given sequence of public inputs will always produce the same public outputs, regardless of the secret inputs. The typical proof method is straightforward: fix a sequence of public inputs. Show that changing the secret inputs does not change the public outputs.

In our setting, let an environment be the mapping of variables to their values. We show that a transition from a given mode and environment will lead to a deterministic next observable mode (Conjecture 1) and a deterministic next environment (Conjecture 2). It remains to be shown that this observations of this next environment are independent from any confidential data. We show that confidential data has no influence on the value of low confidentiality expressions (Conjecture 3). Finally, we show that confidential data has no influence on observations from transitions (Conjecture 4).

5.2 Determinism

First, we show that we have sufficiently restricted non-determinism to avoid the problems described in Section 3.4. Note that not all our expressions are deterministic, because we

Case	**Possibilistic determinism**
Integers	$\overline{\{n\} :: \texttt{int}\ \sigma}$
Booleans	$\overline{\{b\} :: \texttt{bool}\ \sigma}$
Integers	$\overline{\{k\} :: \texttt{key}\ \sigma}$
Tuples	$\dfrac{\hat{u}_1 :: \tau_1 \quad \dots \quad \hat{u}_n :: \tau_n}{\hat{u}_1 \times \dots \times \hat{u}_n :: (\tau_1, \dots, \tau_n)}$
Ciphertext	$\dfrac{\{v \mid u \in \texttt{encrypt}(v,k), u \in \hat{u}\} :: \tau}{\hat{u} :: \texttt{enc}\ \tau\ \sigma}$
Environment	$\dfrac{\{E(x) \mid E \in \hat{E}\} :: T(x)\ \sigma}{\hat{E} :: \texttt{environment}}$

Table 7: Possibilistic determinism

are dealing with possibilistic non-interference: encrypting a value (non-deterministically) yields *any* value in the ciphertext domain. Therefore, we must first formalize what we mean by deterministic. Intuitively, a system is deterministic if the outcome of a transition from one state to another is determined by the input. On LTSs, this 'outcome' is the transition label and the resulting state. The state is determined by the mode and the value of all variables and data ports. Let an *environment* E map variables and data ports to their values. A state in the LTS is then given by (m, E), where m is the current mode and E is the current environment. We first show that the target mode is unique:

CONJECTURE 1. *Let E be an arbitrary environment and m an arbitrary mode. For every pair of transitions $(m, E) \xrightarrow{l_1} (m_1, E_1)$ and $(m, E) \xrightarrow{l_2} (m_2, E_2)$, either $m_1 = m_2$, or l_1 and l_2 are distinct input event ports of the root component.*

Proof sketch: follows from the restriction that events and guards do not overlap.

We then show that the target environments only possibly differ in their ciphertexts. Importantly, when such a ciphertext is again successfully decrypted, the value obtained is again deterministic. We define *possibilistic determinism* on sets of values of a given type as in Table 7. We also define when a set of environments is possibilistically deterministic, which is when for each variable and each possible value for that variable, there is an environment that maps that variable to that value. We now show the possibilistic determinism of our models. Determinism for modes had already been shown. What is left to show is the possibilistic determinism of the environments:

CONJECTURE 2. *Let m be a mode and $\hat{E}$ be a possibilistically deterministic set of environments. The set of target environments $\hat{E}' = \{E' \mid (m, E) \xrightarrow{l} (m, E'), E \in \hat{E}\}$ is possibilistically deterministic.*

Proof sketch: the value of a variable or port can only be changed through the transition's effect, which is an assignment of the form $x := e$. We prove that possibilistic determinism is maintained, by structural induction on the expression e. The most interesting cases are encryption and decryption. For $e = \texttt{encrypt}(v, k)$, the set of possible values for x is trivially the set of ciphertexts generated by v and k, which is exactly the set $\hat{u}$ in the case for ciphertext in the definition of possibilistic determinism. For $e = \texttt{decrypt}(u, k)$, we use that decryption is deterministic: for any ciphertext u generated by encrypting a value v, we re-obtain that value v.

5.3 Expressions

Lifting the definition of low equivalence to sets is straightforward: let $\hat{v}$ and $\hat{v}'$ be two sets of values. These two sets are low equivalent iff. for each $v \in \hat{v}$ there exists a $v' \in \hat{v}'$ that is low equivalent, i.e. $v \sim_\tau v'$, and vice versa; for each $v' \in \hat{v}'$ there exists a $v \in \hat{v}$ s.t. $v \sim_\tau v'$. Similarly, two sets of environments $\hat{E}$ and $\hat{E}'$ are low equivalent iff. for each x and each $E \in \hat{E}$, there exists an environment in $E' \in \hat{E}'$ such that $E(x) \sim_{T(x)} E'(x)$, and vice versa.

Possibilistic non-interference on expressions can now be expressed as follows: if an expression is evaluated in low-equivalent sets of environments, yielding sets of possible values, then these sets are again low-equivalent.

CONJECTURE 3. *Let $T \vdash e : \tau$ and let $\hat{E}, \hat{E}'$ be two low equivalent possibilistically deterministic environments. Let $\hat{v}$ be the set of possible evaluations of expression e in each environment $E \in \hat{E}$, and $\hat{v}'$ be the set of possible evaluations of e in each environment $E' \in \hat{E}'$. Then $\hat{v} \sim_\tau \hat{v}'$.*

Proof sketch: to show is that for each $v \in \hat{v}$ there exists a low equivalent $v' \in \hat{v}'$, and vice versa. Because of symmetry, one direction suffices. The proof is by induction on the derivation of $T \vdash e : \tau$. The interesting cases are encryption and decryption, where we use the properties of encryption and decryption at the end of Section 3.2.

5.4 Specification

Finally, we show that a typeable specification is non-interfering. This follows from possibilistic determinism on the trace and possibilistic non-interference of typeable expressions.

It has already been shown that up to possibilistic determinism, there is only one trace of modes and transitions through the model, for a sequence of input events: for a mode m, at most one transition m `-[`p `when` g `then` $x := e$`]->` m' is enabled per input event port p, or, if p is an output event port or omitted altogether, it is the only enabled transition from m. It is important to note that transitions which provide an observable event or data change in a low confidentiality variable or port, are only enabled from *low confidentiality modes*. Thus, the observable part of this one trace consists of the transitions from a low mode to some other mode.

We show that low equivalence is preserved under transitions:

CONJECTURE 4. *Let m be a mode, $\hat{E}_1$ and $\hat{E}_2$ be two possibilistically deterministic sets of environments, and let m `-[`p `when` g `then` $x := e$`]->`m' be a typeable transition. Let $\hat{E}_1' = \{E_1' \mid (m, E_1) \xrightarrow{p} (m', E_1'), E_1 \in \hat{E}_1\}$ be the set of reachable environments via this transition, and similarly for $\hat{E}_2'$. Then $\hat{E}_1' \sim \hat{E}_2'$.*

Proof sketch: we make a case distinction based on the cases for the transition type rule outlined in Section 4.2. The interesting cases are the ones with an observable effect. The other cases are obviously low equivalent. Moreover, traces along high modes necessarily reach a unique low mode because of the restrictions of Section 3.5 and possibilistic determinism. The cases with an observable effect are where the transition either has a low confidentiality event or makes an assignment to a low confidentiality port or variable. The cases are similar and we sketch the proof for the latter. As x is of low confidentiality, and the type of x is a subtype of that of e, the expression e must also be of low confidentiality. As shown in Conjecture 3, this implies that for two low equivalent environments E_1 and E_2, the possible evaluations of the expression e are also low equivalent. A similar argument applies to the guard expression: for two low equivalent environments E_1 and E_2, the possible evaluations of a low confidentiality guard g are also low equivalent, so the fact that the guard is true (as the transition had been taken) does not distinguish between E_1 and E_2: either the transition is enabled in both, or in neither. This means the sets of possible reachable environments are low equivalent.

6. CONCLUSIONS

In this paper we have investigated an approach to tracking the global information flow in security-critical systems in the presence of cryptographic operations. The analysis is based on an extended notion of non-interference, called possibilistic non-interference, which postulates that the result of encryption is possibly *any* ciphertext. A corresponding analysis has been developed for a MILS variant of the AADL architecture description language. It is given as a type system that ensures possibilistic non-interference properties of a system specification by distinguishing between breaking standard non-interference because of legitimate use of sufficiently strong encryption, or due to an unintended information leak.

As future work, we are planning to elaborate the correctness proof of the type system as sketched in Section 5, and to implement it as part of the D-MILS toolset. We also plan to investigate type inference methods in addition to type checking.

7. REFERENCES

[1] The D-MILS project web site. `http://www.d-mils.org/`.

[2] A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. *Theor. Comput. Sci.*, 402(2-3):82–101, 2008.

[3] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical report, ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975.

[4] K. J. Biba. Integrity considerations for secure computer systems. Technical report, USAF Electronic Systems Division, Bedford, MA,, 1977.

[5] Specification of MILS-AADL. Technical Report D2.1, Version 2.0, D-MILS Project, July 2014.

[6] Intermediate languages and semantics transformations for distributed mils – part 1. Technical Report D3.2, Version 1.2, D-MILS Project, Feb. 2014.

[7] C. Dima, C. Enea, and R. Gramatovici. Nondeterministic nointerference and deducible information flow. Technical report, 2006.

[8] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[9] J. Jacob. Security specifications. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 18-21, 1988*, pages 14–23, 1988.

[10] P. Laud. On the computational soundness of cryptographically masked flows. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 337–348, 2008.

[11] D. McCullough. Noninterference and the composability of security properties. In *Proceedings of the 1988 IEEE conference on Security and privacy*, SP'88, pages 177–186. IEEE Computer Society, 1988.

[12] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report SRI-CSL-92-2, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1992.

[13] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1), 2003.

[14] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*, pages 255–269, 2005.

[15] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[16] Architecture Analysis & Design Language (AADL) (rev. B). SAE Standard AS5506B, International Society of Automotive Engineers, Sept. 2012.

[17] D. M. Volpano. Secure introduction of one-way functions. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, July 3-5, 2000*, pages 246–254, 2000.