RWTH Aachen University
Lehrstuhl für Informatik 2
Software Modeling and Verification

Bachelor Thesis

Analysing Cryptographically-Masked Information Flows Using Slicing

August 22, 2016

Author: Louis Wachtmeister
First Reviewer: Prof. Dr. Thomas Noll
Second Reviewer: Prof. Dr. Joost-Pieter Katoen

Abstract

Whenever the architecture of a system allows untrusted application programs to access sensitive information, a technique must be provided to prevent such information from being "leaked" and becoming available to unauthorised entities. As a result, information flow control mechanisms are introduced, which allow to check the design of programs for security leaks and illegal influences of critical computations based on the system description. However, many information flow control mechanisms are either imprecise, which results in many false alarms, or are unable to handle cryptographic operations.

Therefore, in this thesis a method for information flow control based on slicing is introduced for system descriptions that are specified in a MILS (Multiple Independent Levels of Security) variant of the Abstract Analysis and Design Language (AADL). The introduction itself is split into two parts: First, a slicing method for system descriptions without any cryptographic operations is introduced. Secondly, this basic method is extended such that cryptographically-masked flows can be analysed by taking the "declassifying" effect of cryptography into account.

Erklärung

Ich versichere hiermit, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht in gleicher oder ähnlicher Form zu Prüfungszwecken vorgelegt habe. Alle verwendeten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Louis Wachtmeister
Aachen, den 22. August 2016

Contents

1. Introduction

Whenever the architecture of a computer system allows untrusted applications to access sensitive information, techniques are needed to prevent this information from being "leaked" and becoming available to unauthorised entities. These techniques are discussed under the term of *computer security* and are used to detect covert channels that expose information that should be kept secret. Eliminating such channels requires that the *information flows* between system components of a system architecture are analysed and illegal flows are detected. Many different approaches in this area rely on a notion called *non-interference*, which was first defined by Goguen and Meseguer in [9]. Non-interference demands that changing secret input parameters of a system must not effect the public output of this system. Although this notion of standard non-interference provides satisfactory results for many applications, it is violated in the presence of cryptographic operations. The main challenge in applications using cryptography is to distinguish between legitimate "violations" caused by using (sufficiently strong) encryption mechanisms and unintended information leaks that expose information that should be kept secret.

One possible solutions was presented by Askarov, Hedin and Sabelfeld who suggested to relax the requirement of non-interference and introduced a form of *possibilistic non-interference* for cryptographic operations [2]. In contrast to standard non-interference, possibilistic non-interference regards the sets of *possible* outputs of a system instead of the actual ones. Therefore, non-interference can be recovered for cryptographic operations, if the possible output sequence is unchanged, and the encrypted messages are indistinguishable for any attacker without access to the decryption key. Based on the notion of possibilistic non-interference, type systems have been developed, that can be used to ensure a secure information flow [24]. However, they suffer from some weaknesses such as an insufficient consideration of encryption key distributions and insensitivity to control flow.

Therefore, an alternative approach based on *slicing* is followed in this thesis. In general, slicing describes a form of static analysis that can be used to determine (potential) dependencies between the inputs and outputs of system components. As a result, slicing can be used for various applications, e. g. to analyse information flows [11] or for improving the efficiency of model checking [18].

To develop this slicing based approach for cryptographically-masked flows, in this thesis a variant of the AADL (*Architecture Analysis & Design Language*) called MILS-AADL [6] is used as system description mechanism. To maintain a reasonable effort, the MILS-AADL descriptions are restricted to a core subset that is presented in [24]. Starting from this system description, a static analysis based on slicing is introduced in order to determine which (public) output depends on which (secret) input, in order to indicate information leaks. After introducing a general approach without cryptographic operations, this analysis is extended such that the declassifying effect of cryptography and the restoring of dependencies by decryption is taken into account. This is done by analysing which encryption keys are accessible in which system component, and which data can possibly be encrypted.

To discuss these points, the thesis is structured as follows. First of all, the necessary background information is provided. Therefore, Chapter 2 introduces important principles regarding computer security and explains the already mentioned concepts, information flow analysis and control, type systems and non-interference in a more detailed way.

Thereafter, in Chapter 3 a core subset of the MILS-AADL language is introduced and modified such that it meets the needs of this thesis. In detail, the abstract syntax and the semantics of this core subset is presented and visualised by introducing a system description of a cryptographic controller.

After discussing the abstract syntax and semantics of the system description mechanism, in Chapter 4 a slicing mechanism for MILS-AADL descriptions is presented and a connection with a modified notion of non-interference is drawn. Based on this result, a method to describe and propagate security levels of the input and output ports is introduced. Finally, Chapter 5 is concluded by a confidentiality checking algorithm for systems without cryptographic operations, which allows to analyse whether a leak can occur or not using the slicing algorithm.

In Chapter 5 the results for security levels and confidentiality checking are extended such that the declassifying effect of cryptography and the restoring of dependencies by decryption is taken into account. Additionally, the concept of declassification is discussed in the context of possibilistic non-interference.

To visualise the concepts developed in this thesis, in Chapter 6 the algorithm for confidentiality checking is applied to some case studies. Finally, Chapter 7 concludes this thesis.

2. Preliminaries

Since a static analysis based on slicing is developed in this thesis, a short overview on the necessary background information is provided in this section. To analyse and control the information flow in a security critical system, the concept of information flow control, mathematical lattices in the function of flow-policy descriptions and the main principles to program analysis are introduced.

2.1. Information Flow Control

Handling and transmitting sensitive data in software systems demands a mechanism to ensure that involved system components maintain the *confidentiality* of the data. To maintain confidentiality, this mechanism must guarantee that confidential data cannot leak to public output ports. The standard idea is to protect sensitive data by restricting the data access to trustworthy components [20]. Unfortunately, it is unrealistic to assume that every component in a large software system is trustworthy and will never leak information to other components by error or malicious behaviour. Consequently, techniques to verify the trustworthiness of a component become essential to prevent unintended information leaks, even under consideration of *implicit information flows*, that are not covered in a classical access control.

One of the common techniques to specify end-to-end confidentiality requirements is to analyse the (possible) information flow, in order to detect illegal flows. During this analysis *information flow-policies* are used to express the allowed information flow between objects. Usually, this is done by introducing *security classes*, to which objects are bound by a *binding method*, and *flow relations*, which defines the allowed information flow [7, 8, 20]. The information flow-policies are then enforced by *information flow controls*, which detect information flows to locations, where the information flow-policy is violated.

A different task information flow control can follow is to guarantee *integrity*. Integrity in this case means that critical computations should not be manipulated from public accessible input ports [11]. While maintaining confidentiality requires that information is prevented from flowing to inappropriate destinations or outputs, maintaining integrity requires that information is prevented from flowing from inappropriate sources or inputs. It is possible to treat integrity dual to confidentiality and therefore enforce it by controlling information flows [4, 20].

In this thesis the focus lies on a method for a language-based information flow control. This means that a textual description of a system, in this case a system description in a system description language, is analysed to detect information leaks, while physical side channels are not considered.

2.2. Lattices

To model security levels and the allowed information flows Dorothy Denning introduced in [7] a *lattice* model for secure information flow. This model allows to define relations between different security classes in order to express the allowed information flow. The underlying mathematical model is the universally bounded lattice, which can be defined as follows.

First of all, the general lattice is defined, which can be extended to a universally bounded lattice in a second step.

Definition 2.2.1 (Lattice)**.** *An algebraic structure $(L, \sqcup, \sqcap)$, consisting of a non-empty set L and two binary operations $\sqcup, \sqcap : L \times L \to L$ is called a lattice, if the following properties are fulfilled for all $a, b, c \in L$*

- *Commutative property: $a \sqcup b = b \sqcup a$ and $a \sqcap b = b \sqcap a$*
- *Associative property: $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$ and $(a \sqcap b) \sqcap c = a \sqcup (b \sqcap c)$*
- *Absorption law: $(a \sqcup b) \sqcap a = a$ and $(a \sqcap b) \sqcup a = a$*

The operations $\sqcup$ and $\sqcap$ are called *least upper bound* and *greatest lower bound*. Therefore, every structure guaranteeing that every subset has a least upper bound and a greatest lower bound can be called a lattice. Using this definition, it is possible to connect every lattice structure $(L, \sqcup, \sqcap)$ with a reflexive, antisymmetric and transitive order relation $\sqsubseteq$ on L as follows. For each element $a, b \in L$

- $a \sqsubseteq b \Leftrightarrow a \sqcup b = b$, because b is the least upper bound of a and b in this relation.
- $a \sqsubseteq b \Leftrightarrow a \sqcap b = a$, because a is the greatest lower bound of a and b in this relation.

Using this connection, a universally bounded lattice can be defined.

Definition 2.2.2 (Universally Bounded Lattice)**.** *A structure $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is called a universally bounded lattice, if it consists of the following elements*

- *A finite partially ordered set L, with the partial ordering $\sqsubseteq$,*
- *A least upper bound operator $\sqcup$*
- *A greatest lower bound operator $\sqcap$*
- *A minimal element $\bot$*
- *A maximal element $\top$*

Note that in some definitions of universally bounded lattices the minimal element $\bot$ and the maximal element $\top$ are omitted, because their existence follows from the finiteness of L. However, to simplify the definition for the following chapters, it is assumed that the maximal and minimal element are defined in the considered lattices.

A flow-policy as defined in [7] is represented as $\langle SC, \sqsubseteq \rangle$, where SC is a (finite) set of security classes and $\sqsubseteq$ is a flow relation expressing the allowed information flow between the security classes. Dorothy Denning proved that a flow-policy forms a lattice, if the flow relation is defined as a reflexive, antisymmetric and transitive relation and $\sqcup$ is a least upper bound operator [7]. Therefore, information flow-policies can be specified by a lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ consisting of the security levels L.

Example 2.2.1. One of the simplest lattices to imagine is the lattice $\mathcal{L} = (\{L, H\}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ that consists of only two points. These points represent for instance the security classes *low* (L) and *high* (H) with $L \sqsubseteq H$. The only allowed information flow is from the lower security class L to the higher security class H, which is represented as $L \sqsubseteq H$ in the flow model. In this lattice L is the minimal element $L = \bot$ and H is the maximal element $H = \top$. Therefore, we get:

- $L \sqcup H \equiv H$
- $L \sqcap H \equiv L$

This example lattice is the basic lattice description of the security levels used in the following examples of this thesis. The notion of *low* and *high* is here just exemplary for any lattice used as security policy with a lower class *low* (L) or a higher class *high* (H). Therefore, by referring public and private, unclassified and classified or public and confidential this lattice is meant, but the explicit name of the security classes has been changed. Principally, it is possible to express more complex lattices. For example the universal lattice for a simple while language developed by Hunt and Sands allows to express the program dependencies by taking the powerset of all program variables as a lattice description [12].

2.3. Non-Interference

A common notion in security related information flow analysis is the notion of non-interference. Non-interference was first introduced by Goguen and Meseguer in [9] and focusses on observable behaviour instead of the source code or description of a computer system. The notion of non-interference requires that changes in confidential data must not affect the observable behaviour of system elements that can be accessed by users or objects without the necessary security clearance. In case of data flows this notion means that confidential data should be prevented from flowing to places where the defined security policy is violated [20]. In the context of the security policy defined in Example 2.2.1 this means that a system is secure if it guarantees that changes of high classified variables or system ports must not cause an publicly observable behaviour of the system. Typically non-interference is formalised by low-equivalence relations $\sim_L$ that have to be preserved if high-values are changed. For the two levels *low* and *high* this can be defined as

$$s_1 \sim_L s_2 \Rightarrow s_1' \sim_L s_2'$$

where s_1, s_2 are two initial system states and s_1', s_2' are two final states after an input value changed or an event changing the state occured. And the low equivalence of the resulting states s_1' and s_2' means that all variables classified as L must coincide with their definition in the initial state. As a result, variations in the high input variables have no effect on low output variables, because they have to be low equivalent after the high input variables (or events) are changed. Therefore, a system providing non-interference assures confidentiality. Additionally, Goguen and Meseguer provided in [10] an unwinding theorem, which allow a simplified proof of non-interference by reducing non-interference to several conditions.

Apart from the notion of standard or Goguen-Meseguer non-interference, many different variations have been developed that either extend the notion of non-interference

or introducing forms of non-interference based on PER relations [21]. For example a transitive non-interference policy is described in [14], [19] to introduce transitive interference and non-interference relations that describe which data elements of which security domains are allowed to interfere. To extend them for cryptographic operations this notion was extended in the same works in order to describe intransitive interference and non-interference relations that are suitable to describe security domains that may influence each other and places where this policy is allowed to be broken by intransitivity. This transitive notion of non-interference is used in [11] to connect sliced program dependency graphs with Goguen-Meseguer non-interference.

A different approach are possibilistic non-interference definitions. Two common notions are *generalized non-interference* [17] and *restrictiveness* or *hook-up security* [16, 17]. Both notions were originally introduced for non-deterministic systems, in which not only a single output for a single input is possible but also a set of possible outputs for single inputs can be given. In this context, generalized non-interference implies that the input of a system may not alter the possible outputs of a system, while restrictiveness implies that the system should respond the same to an input independently of changes to higher classified inputs that are changed immediately before [17]. In addition to these advantages concerning non-deterministic systems, a possibilistic non-interference definition allows a refined argumentation when it comes to the discussion of cryptographically-masked flows [2]. As for Goguen-Meseguer non-interference, unwinding theorems are provided for various forms of possibilistic non-interference definitions in [13] but are not discussed in this thesis.

Based on this general overview, different forms of non-interference are discussed in the context of slicing and cryptographically-masked flows in Chapter 4.3 and Chapter 5.5. However, this thesis mainly focusses on the general aspects and ideas of non-interference, while detailed proofs based on unwinding theorems are not provided.

2.4. Principles of Program Analysis

In the literature on Language-Based Information-Flow Security [20] and Flow-Sensitive, Context-Sensitive and Object-Sensitive Information Flow Control [11] the following principles and properties are used to describe the information flow control and program analysis methods. Any program analysis is subject to several conflicting requirements:

- *Correctness* is an essential property of any information flow control analysis. It describes that the analysis must be able to find any potential security leak that is present in the system description.

- *Precision* means that there are as few false alarms as possible. From this property follows that the most of the programs condemned by the information flow control analysis must be insecure.

- *Scalability* demands that the analysis is able to handle realistic programs with a reasonable effort.

- *Practicability* demands that the analysis must be usable with a reasonable effort. It implies that descriptions for the result of an analysis are provided and understandable. Additionally, using the analysis must not require a high effort from the user.

Unfortunately, achieving all properties at the same time is impossible, because total precision cannot be achieved while maintaining correctness [11]. Additionally, a higher precision influences the scalability in a negative way, which leads to the problem that a precise analysis requires a higher effort while a fast analysis leads to many false alarms [11]. As a result, it is necessary to follow a *conservative approximation*, where a few false alarms are allowed but all security leaks are detected, to achieve correctness. In addition, it is important to find a good trade-off between scalability and practicability by developing fast algorithms that allow only a few false alarms.

2.4.1. Sensitivity

As a method to choose between the effort and precision of an analysis, the analysis is often characterised by the following properties [11]:

- *Flow-Sensitivity* describes that the order of statement execution is taken into account. Consequently, the analysis of single statements might lead to a different result than the analysis of the complete system, because insecure sub-statements become secure e. g. by changing the output variable to non-sensitive content by another statement. On the other hand, a *flow-insensitive* analysis does not take the order of execution into account. This means that every sub-statement must be secure, which might result in false alarms.

- *Context-Sensitivity* means that the procedure calling context is taken into account. This results in a computation of separate information for different calls of the same procedure. In contrast to the context-sensitive analysis in a *context-insensitive* analysis all call sides of a procedure are merged.

- *Object-Sensitivity* describes that different objects for the same field (attribute) of an object are taken into account, whereas in an *object-insensitive* analysis the information for a field over all objects is merged to one class.

In general, sensitive analyses are more precise than insensitive ones. However, the scalability of the process is influenced in a negative way, because sensitive analyses are more expensive than insensitive ones. As this thesis focusses on system descriptions and not on object oriented programming languages, the main focus in this thesis lies on flow-(in)sensitivity.

2.4.2. Covert Channels

Channels are mechanisms used to signal information through a computer system. Information that is signalled through a channel from the output of one system component to another system component's input causes *explicit information flows*. Information leakage caused by these explicit flows are easy to imagine, because here information is transferred directly through channels that are meant to signal information. More complicated are information transfers caused by mechanisms that are not meant for information transfer, called *covert channels*. In [20] Sebelfeld and Myers categorise them as follows:

- *Implicit flows* signal information through the control structure of a program. E. g. conditions might indicate characteristics of objects through different return values.

- *Termination channels* signal information through the termination behaviour of a system. For example, an attacker might follow from the non-termination of a system which loop condition or execution path was chosen.

- *Timing channels* signal information through the time it takes until an action (e. g. program termination) occurs. Different program execution paths may cause different computation times. Therefore, it might be possible for an attacker to guess which program path was executed and therefore access sensitive information, for example by guessing whether a condition is fulfilled or not or he might guess whether an entry is part of a database or not by comparing different search times.

- *Probabilistic channels* signal information by influencing the probability distribution of observable outputs. Repeatedly running a computation while observing the stochastic properties might result in information leaks.

- *Resource exhaustion channels* signal information by the consumption of shared resource, such as memory space or CPU usage.

- *Power channels* signal information through the power consumption of components used for the computation.

Which covert channel is problematic for the security of a system highly depends on the information an attacker can access. For example, power channels and resource exhaustion channels can only leak information to attackers that have access to information that reveals the power consumption or the consumption of finite shared resources. Attackers without access to this information cannot hope to use one of these covert channels to get sensitive information.

In this thesis the main focus lies on analysing the data flows between system components based on the architecture of the system. Therefore, the consideration of implicit flows caused by the control structure of a system is the most important and therefore the main focus lies on this channel. Additionally, we claim that attackers have no hardware access and therefore no access to power or resource consumption indicators.

2.5. Security Type Systems

Security type systems can be used to enforce information flow-policies [20]. This is done by attaching *security levels* to objects, in order to relate them to classes of the information flow-policy. Introducing *type rules* allows to enforce confidentiality by propagating the security levels through the statements, while guaranteeing to catch illegal information flows [11]. The security levels are either bound statically, which results in a constant type for each object, or bound dynamically, which allows the security level of an object to vary depending on its content. The binding method (static binding or dynamic binding) influences the flow-sensitivity of the analysis, hence statically bound security types result in a flow-insensitive analysis, whereas dynamically bound security types may allow a flow-sensitive analysis depending on the used type rules [12]. The type rules are derivation rules of the form $pc \vdash C$, which means that the program C is typeable in the security context pc. Additionally, derivation rules of the form $pc \vdash exp : \tau$ are introduced to mean that an expression

exp has type τ under consideration of the security context *pc*. The idea behind this security context *pc* is to rule out implicit information flows by using it as a program counter level, which tracks the dependencies of a program counter to ensure that only variables greater than this program counter are changed by the program C.

Type systems have many advantages: The type based information flow control is relatively efficient and the structure of type rules aids in a rigorous proof of its correctness [12]. However, they can be rather imprecise, as many of them are neither flow-sensitive, context-sensitive nor object-sensitive, such as the type system for cryptographically-masked information flows used for security type checking in MILS-AADL specifications presented in [24]. There are type systems that are flow-sensitive like the type system by Hunt and Sands [12], but most of them are only suited for simple while-languages.

2.6. An Overview on Related Works

One of the first information flow control mechanisms is the static program analysis developed by Denning and Denning [8]. This analysis is used to determine, whether the information flow properties of a specific program satisfy a confidentiality police or not. Later, Volpano et. al. extended this analysis to a type system and showed that programs typeable in their type system are secure according to the notion of non-interference [25]. Non-interference demands that changing secret input parameters must not effect the value of the public output parameters of a system [9]. A general overview on type systems and other language based approaches is given in [20]. Based on the flow-insensitive type system by Volpano et. al. Hunt and Sands developed a flow-sensitive approach [12], which is dual to the analysis of program variable independence by Amtoft and Banerjee who use a Hoare-like logic [1]. However, these approaches are not suited for a practical use or a system analysis, because they are based on simple while languages. Therefore, Hammer et. al. introduced a flow-sensitive, context-sensitive an object-sensitive information flow control mechanism based on program dependency graphs (for Java Bytecode) [11]. The results on slicing and declassification are summarised in Chapter 5. Despite of the differences between the flow-sensitive type system developed by Hunt and Sands and the slicing approach developed by Hammer et. al., Heiko Mantel and Henning Sudbrock showed in [15] that type based and slicing based information flow analyses must not differ in their precision.

However, these approaches all suffer from the weakness that encrypted information flows or data that is released after statistical anonymisation cannot be handled, because the non-interference criterion is to strong. To solve these problems different approaches for declassification were introduced, which allow data elements to be released under specific conditions. A general overview on the dimensions and principles for declassification are given in [22] and summarised in this thesis in Chapter 5.1. To handle cryptographic operations Askarov, Hedin and Sabelfeld introduced a method to analyse cryptographically-masked flows, using the notion of possibilistic non-interference [2]. This was extended to a type system for MILS-AADL Specifications in [24].

The approach developed in this thesis is related to the type system developed in [24], as the thesis aims to introduce a flow-sensitive information flow control based on slicing. The main ideas of the slicing approach are inspired by [11] and based on the

slicing approach introduced in [18].

3. The MILS-AADL Language

Since determining, whether (public) output ports of a given system description depend on (secret) input ports, is one of the main tasks of the analysis developed in this thesis, a suitable system description formalism is needed. Therefore, a MILS (*Multiple Independent Levels of Security*) variant of the AADL language, called MILS-AADL [6], is used to describe these systems. This specification language has been developed within the D-MILS (*Distributed-MILS*) project and is intended to present the model-based design of D-MILS systems to the user. It has the following key features:

- Considered systems are hierarchically organized into components.
- The component architecture and its interaction is expressed by connections between data and event ports.
- The internal behaviour is described.
- Security-related mechanism such as encryption and authentication are provided.

3.1. Syntax

Instead of the full MILS-AADL language, which is presented in [6], in this thesis a simplified version of the language is used. The syntax and semantics of this language are based on the core subset of MILS-AADL Kevin van der Pol and Thomas Noll used in [24] as system description mechanism for a type system based analysis for cryptographically-masked information flows. The general syntax of this core subset of MILS-AADL can be found in Table 3.1. The main difference to the core subset presented in [24] is in the additional consideration of data flows and a different method for introducing keys and their distribution. In Table 3.1 n is a numerical value, x is a variable name, k a named key pair, p_e an event port, p_d a data port, s the name of a system and m a mode name.

One of the advantages of MILS-AADL is the combination of the architecture description and the behavioural description of a system. The architecture of the system is described by specifying *hierarchies* of (sub)systems and *connections* between their input and output ports. The hierarchical description of different system components is recursive, which means that each of the described subsystems may also contain hierarchically ordered subsystems, connected by input and output ports. To define a starting point, the outermost or highest system in this hierarchy is called the *root system*. The allowed communication of system components is described by *event ports* and *data ports*. Event ports are able to trigger changes in the behaviour of a system, whereas data ports are used to transmit or receive data values to or from the environment. Connections between ports can be distinguished into *event port connections* and *data flows*. Event port connections describe connections between event ports, whereas connections between data ports are called data flows. A connection between event and data ports is not intended and therefore not provided.

Case	Grammar
Type	$\tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{enc}\ \tau \mid (\tau, \ldots, \tau)$
Key Pair	$\kappa ::= k_p : \texttt{key pair}$
Key	$K ::= k : \texttt{key pub}(k_p) \mid \texttt{key priv}(k_p) \mid$
Expression	$e ::= n \mid x \mid e \oplus e \mid (e, \ldots, e) \mid e[n]$
System	$S ::= \texttt{system}\ s(S^*\ P^*\ C^*\ V^*\ M^*\ T^*)$
Event Port	$P_E ::= p_e : (\texttt{in} \mid \texttt{out})(\texttt{event})$
Data Port	$P_D ::= p_d : (\texttt{in} \mid \texttt{out})(\texttt{data}\ \tau\ e)$
Port	$P ::= P_E \mid P_D$
Event Port Connection	$C ::= \mathit{connection}([s.]p_e, [s.]p_e)$
Variable	$V ::= x : \tau\ e$
Data Flow	$F ::= \mathit{flow}([s.]p_d\texttt{->}[s.]p_d)$
Mode	$M ::= m : [\texttt{initial}]\ \texttt{mode}$
Transition	$T ::= m\ \texttt{-[}[p]\ [\texttt{when}\ e]\ [\texttt{then}\ x := e]\texttt{->}\ m'$

Table 3.1.: MILS-AADL syntax

The relevant behaviour of the system is the communication between different subsystems of the root system via system ports. This system behaviour becomes observable by considering the input and output ports of a system. To model the possible communication behaviour of a system, an automaton with *modes* and *transitions* is introduced. In this description model, modes represent the current system state, whereas transitions serve to change the system state and to describe the behaviour of the system in this process. The transitions of the description automaton can be labelled with the following elements in order to model the requirements that are necessary to change the mode and the effect taking this transition has to the output ports.

- The *event* is given by an event port p and describes the event, which is consumed or produced, depending on the port definition (input/output-port). If this label is omitted, no port event will be consumed or produced.

- The *guard* expression restricts the transitions that can be taken at a specific system state, by only allowing transitions to be taken, whose guard expression evaluate to true. Omitted guard expressions are considered as true.

- In the full version of MILS-AADL a list of *effects* $x_1 := e_1; \ldots; x_n := e_n$ describes the assignments that result from taking the transition. In this process, expressions are evaluated in the source mode and assigned simultaneously to these variables. An omitted effect results in unchanged variables. To simplify the analysis developed in this thesis, only one effect per transition will be allowed. Therefore, to realise multiple effects, new modes and transitions must be added to perform each effect in a single transition.

According to the specification of full MILS-AADL it is necessary to define default values for the data ports of subcomponents except those of type (`private` or `public`) `key` with matching data types (H-5 - H-7) [6, p. 24]. To define these default values the definition of data ports $P_D ::= (\texttt{in} \mid \texttt{out})\texttt{data}\ \tau\ e$ the expression e is used additionally to the type τ.

With respect to cryptographic operations and security, MILS-AADL provides *security primitives* in its expression language. These primitives allow to encrypt or sign values by asymmetric or symmetric encryption and authentication mechanisms. In this thesis only asymmetric cryptography is considered. This is possible, because in the context of confidentiality analysis, symmetric cryptographic operations can be treated the same as asymmetric cryptography by classifying the public keys as confidential and set them to the value of the private keys. To declare key pairs, global constants of type κ are defined on top level. The keys itself are defined as private or public subkeys in a subcomponent. This definition allows to define a static key pool on the one hand, while a dynamic distribution of the keys is guaranteed on the other hand. For a key pair κ a corresponding public key k and private key k' can be used for cryptographic operations as follows. The encrypt function `encrypt`(m, k) takes a message m of type τ and a public key k : `key` as input and returns a ciphertext with type `enc` τ. To decrypt the ciphertexts the function `decrypt`$(c, k') : \tau$ is introduced. This function takes a ciphertext c : `enc` τ and a private key k' : `key` and returns the original message m, if k' is the corresponding private key to the public key k used for encryption. The function leads to a deadlock in the statement that contains the decryption expression, if k' does not match to k. Since a secure forwarding of encrypted messages becomes utterly impossible, if encrypted messages are readable for any attacker without knowing the private key, it is inevitable to assume that a message encrypted with a senders public key can only be decrypted by possessors of the matching private key. To simplify the method developed in this thesis it is assumed that only one encryption operation is allowed per transition. This means that transitions of the form `m -[then x:= encrypt(encrypt(y,k),k')]-> m'` must be adapted such that an intermediate mode and a unused variable is introduced such that this transition can be split into two encryption steps. This reformulation results in the transitions `m -[then z:= encrypt(y,k)]-> n` and `n -[then x:= encrypt(z,k')]-> m'` where `n` is a new mode and `z` a unused variable. The same principle can be applied for decryption.

As in [24] the other security primitives, signing and hashing, that are provided in full MILS-AADL are not discussed in this thesis. For signing, this is because signing can be discussed under the term of integrity, which can be seen as fully dual to confidentiality (see Section 2.1). Therefore, an analysis that allows to check for confidentiality can also be used to check for integrity after minor modifications. For hashing, this is because information can be leaked by guessing the content of a hashed value and check whether its guess was correct. Even under consideration that this guess will be incorrect in most cases, which makes hashing secure in quantitative settings, this information leak is generally possible and therefore hashing is not secure in a qualitative setting.

Example 3.1.1. As an example, we want to consider a modified cryptographic controller system, which is presented in its original form in [24]. The cryptographic controller gets a low classified header and a high classified payload as input. In the next step, the header is transmitted to a merge component unchanged by a bypass component, whereas the payload is first encrypted and then transmitted to the merge component. The header and the encrypted payload, which is now considered as low, are merged in the merge component and released to a public output. This cryptographic controller system can be used to encrypt confidential data before releasing it to public outputs, to maintain confidentiality of the payload.

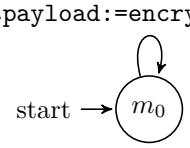

Figure 3.1.: The automaton describing the crypto component

```
system cryptocontroller(
 header: in data int 0
 payload: in data int 0
 outframe: out data (int,enc int) (0,encrypt(0, k0))
 mykeys: key pair
 m0: initial mode
 system bypass(
  inheader: in data int 0
  outheader: 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: keay pub(mykeys)
  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
 )
 flow header -> bypass.inheader
 flow payload -> crypto.inpayload
 flow bypass.outheader -> merge.header
 flow crypto.outheader -> merge.payload
 flow merge.frame -> outframe
)
```

In this example the root system *cryptocontroller* is hierarchically ordered into the subsystems bypass, crypto and merge. To visualize the modes we represent them as circles and transitions as arrows between the circles. The arrows are labelled with event, guard and effects (if not omitted). As an example the crypto systems behaviour is represented in the automata in Figure 3.1. All ports with type `int` of this cryptographic controller component have the default value 0. The default key is defined as k_0.

3.2. Semantics

Instead of the semantics of full MILS-AADL, which are defined in [5], in this work the presented semantics are directly reduced to *labelled transition systems* (LTS), which

is done analogous to [24]. In this reduced version each component has its own LTS to describe its behaviour. The states of the labelled transition system are given by the modes of all systems, their subsystems and the values of the data ports and variables [24]. Formally this state space S is defined by the cartesian product of the states and the possible values of the data ports. The initial state of this system S_0 is defined by the initial mode m_0 of the considered component and the default values of the data ports. Transitions of the LTS are given by the transitions between these modes and by changes in input data ports and variables. The transitions can either be internal or external. External transitions are labelled with the corresponding event, whereas internal transitions are unlabelled. To respect the transition guards, mode transitions are only enabled in the LTS if and only if the guard enables to true and at least one of the following conditions is fulfilled.

1. There is no event. In this case the transition is an internal transition, which can always be activated, if the guard expression evaluates to true.
2. The event is an input event port and the system is the root system. In this case, the input event comes from the environment of the root system.
3. The event is an input event port, which is connected to another systems output event port and both involved systems simultaneously take a transition on this event. In these cases two connected systems influence each other by their input/output-behaviour.
4. The event is an output port, which is connected to another systems input port. In this case also two connected systems influence each other.

As a result, all other events that do not satisfy the guard expression or one of the conditions above are ignored by the LTS. To model the influence of data ports, input data ports are considered to be controlled by the environment and can change their value at any time, whereas output data ports only change their value in an effect. Input events ports provide events that are able to trigger the transition labelled with this event. Output event ports emit such events whenever a transition is taken that has this port as event.

Connections between data ports are considered to transfer data instantaneously. This is also modelled in the transition that changes the data value in the source end. The same applies for event port connections.

Example 3.2.1. The semantics of the crypto component of the cryptographic controller presented in Example 3.1.1 can be described by the following LTS. The state space is defined as the cartesian product of the mode and the possible input and output values, as well ass all possible variable assignments. In this example, the carthesian product describing the state space is the carthesian product of the singleton state $\{m_0\}$, the integer values i and o for the values of the input data port `inheader` and `outheader` and the keys k that are used for encryption. As a simplification, the keys are considered to be taken from the set of keys $\mathbb{K}$. Mathematically the set of states S can be described as:

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

Additionally, the labelled transition system has an initial mode, which can be defined as the initial mode of the component in combination with the default values in

the declaration. In this example this leads to the following initial state s_0:

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

There are two possible cases for transitions whose union can be defined as the transition relation $\rightarrow$. In the first case, only the input values change, whereas all other values stay the same. This case leads to transitions of the form

$$(m, i, o, k) \xrightarrow{i,i'} (m, i', o, k) \tag{3.1}$$

for mode a m, input value i, new input value i', output o and key value k. In the second case, a transition is taken spontaneously, which would lead to an update of the output port o to the encryption of the input value i. This case leads to transitions of the form

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

4. Slicing AADL Specifications and Non-Interference

To control the information flow of programs Hammer and Snelting introduced in [11] an information flow control based on slicing. In contrast to type systems, information flow controls based on slicing have the advantage that they are inherently flow-sensitive. However, their approach is based on program dependency graphs (PDG) and not intended to be used for MILS-AADL specifications. Therefore, a different slicing algorithm is extended in this thesis such that it can be used as information flow control for MILS-AADL specifications. The main slicing algorithm used in this thesis was developed by Odenbrett, Nguyen and Noll [18] and is originally used for model-checking of AADL-Specifications. In this chapter this basic slicing algorithm is extended such that it can be used for the core subset of MILS-AADL defined in Chapter 3. Additionally, the concept of slicing is connected to the notion of non-interference, which is used to describe that changing secret input parameters must not effect the value of public output parameters. Since standard non-interference is often connected with a definition of different security levels, functions to model security levels are introduced, which maintain non-interference. However, we will see that this method based on the security concept non-interference is not able to handle cryptographic operations. Therefore, this approach is again extended in Chapter 5 such that it can handle cryptographically-masked flows.

4.1. Slicing MILS-AADL Specifications

Odenbrett, Nguyen and Noll introduced in [18] a slicing algorithm for AADL specifications that can be used for model checking. For a given specification S and a slicing criterion φ the algorithm reduces the original specification to a smaller specification S^{φ}_{sliced}. The principle idea is thereby to reduce the original system description S to the (potentially) smaller system description S^{φ}_{sliced}, by only maintaining the parts of the system description influencing the slicing criterion φ. For information flow control this property can be used, because the resulting system S^{φ}_{sliced} must only contain elements influencing the slicing criterion. Therefore, the information flow to the output ports can be analysed by slicing the system specification S with respect to the output ports. In the resulting system, all maintaining parts of the system description have a direct influence on the interesting port and thus can be used for a further analysis respecting security levels or non-interference. This consideration is important as outputs with a low security level should not depend on inputs having a higher security level.

In order to use the algorithm presented in [18] as slicing algorithm for information flow control, the initial values must be varied slightly. Instead of a property φ, the sets I_D and I_E of interesting data elements and events are used as slicing criterion. In addition to these sets of data elements or events, a set of modes is used, which

is assumed to be initially empty. The slicing criterion serves to describe the initially interesting parts of a specification that must not be sliced away by the algorithm. Therefore, the set I_D must contain the initially interesting data elements and the set I_E must contain the initially interesting events and finally I_M contains the interesting modes. A slicing for mode dependencies is not intended, and therefore the set containing the initial modes is assumed to be the empty set in most applications. Starting from these initial sets, all other aspects that have a direct or indirect influence on them are added. The step before is iterated until a fixpoint is reached in order to get all dependences. The termination of this algorithm is obviously guaranteed for finite MILS-AADL specifications but in the worst case all parts of the program become interesting. Interestingness is thereby defined as influencing the elements in the slicing criterion. The special aspects to identify interesting elements and a more detailed overview on the closure rules developed in [18] are provided in the following paragraphs.

- *Identifying Interesting Data Elements:* To cover explicit data flows all data elements that are used to calculate interesting data elements have to be considered as interesting, too. In case of assignments within transition effects, this is the right hand side of the assignment. In case of data flows, the source of an interesting element must also be considered as interesting element. In addition to these explicit flows, also the implicit information flows caused by the control structure of the specification have to be considered. Therefore, data elements that are used in the transition guard have to be kept in the sliced specification, because evaluating this expression determines if the transition can be taken.

- *Identifying Interesting Events:* Unlike sequential programs, the components of a MILS-AADL specifications are able to synchronously communicate by sending and receiving events. Regarding the possible forwarding of events via this bidirectional event communication, all events connected to interesting events in any direction must be considered as interesting, too.

- *Identifying Interesting Modes:* The algorithm presented in [18] is not designed to identify interesting modes, but is focussed on the elimination of uninteresting modes. Most of the aspects analysed for (un)interestingness are based on the transition relation, which is not considered directly in the slicing algorithm. For example, interesting transitions change the values of interesting data elements, have interesting triggers, or lead into interesting modes. Furthermore, transitions with interesting triggers, guards and source modes have to be considered as interesting. To take this important property into account without regarding these interesting transitions alone, modes are made interesting and the in and outgoing transitions become indirectly interesting, too. This is done by regarding modes with one of the following properties in the iteration.
  - Source modes of transitions, which change interesting data elements.
  - Modes that are active in data flows or an interesting event port connection.
  - Source modes of transitions with interesting events as triggers.

  To respect the reachability, every predecessor of an interesting mode has to be considered as interesting, too.

After introducing the term of interestingness it is necessary to define, which effects are interesting with respect to information flow control. Resulting from the definition

of a information flow-policy, data leaks can only be caused by output ports that are influenced by elements with a higher security clearance. Therefore, the initially interesting data elements should either be the data elements that are released to public output ports, or data elements that are considered as secret and not meant to be released to unauthorised entities. Releasing information in this case means that information is signalled by an output port which allows to deduce aspects of the information or the information itself. Based on this idea in Section 4.4.2 a theorem is provided, which allows to combine the dependencies found by a slicing algorithm with the security levels of data elements and event ports. In addition to security leaks revealing any information about data elements or event triggers, it is allowed to define a set of initially interesting modes to simplify the flow equations introduced in chapter 4.4.

4.2. The Slicing Algorithm

The modified slicing algorithm based on the algorithm from [18] is presented in Algorithm 4.1. The set Dat is used to describe data elements, Evt describes a set of events and Mod is a set of modes that occur in the specification S. To model transitions the relation Trn contains a set of transitions of the form $m \xrightarrow{e,g,f} m'$ for mode $m, m' \in Mod$, a trigger $e \in Evt$, a guard expression g over data elements and a list of assignments f. Data flows of the form $a \rightsquigarrow d$ are collected in the set Flw. Data flows in this case are either flow connections between data ports or direct assignments. In both cases, they are described by the list of effects in the transition. Finally, connections between event ports denoted as $e \rightsquigarrow e'$ for two event ports $e, e' \in Evt$ are collected in the set Con. The result of the algorithm are the three sets D, E, M, where D contains the interesting data elements, E contains the interesting events and M contains the interesting modes. This means that for confidentiality three sets I_D, I_E and I_M of interesting data elements, events and modes are the input of the algorithm, whereas the final sets D, E and M contain all data elements, events and modes that may influence these interesting data elements.

Algorithm 4.1 calculates the so called backward slice. It is called backward, because all modes, events and data elements that are tracked in this analysis are calculated starting from the resulting element going backward in the system description. During an information flow analysis based on slicing, the order of statements has an influence on the result, because the starting point of S is given and the dependencies are tracked backwards step by step. Therefore, an information flow control based on slicing is flow-sensitive.

To simplify the definitions made in the following sections, the functions $D(x), E(x)$ and $M(x)$ are introduced. For a singleton set $\{x\}$ the resulting sets contain the data elements, events or modes generated by Algorithm 4.1 using $\{x\}$ as initial set for data elements, events or modes depending on the type of x.

4.3. Non-Interference and Slicing

In [11] and [23] Snelting et. al. mention that for program dependency graphs, a node a satisfies the Goguen-Meseguer non-interference criterion, if the security domains of all elements c in the backward slice of a are allowed to interfere with the security domain of a. The definition of allowed and disallowed interference is thereby based on

Algorithm 4.1 The Slicing Algorithm

```
procedure calculateBackwardSlice(S, I_D, I_E, I_M)
input:
  S: A System description
  I_D: The initially interesting Data Elements
  I_E: The initially interesting Events
  I_M: The initially interesting Modes
output:
  D: All Data Elements depending on I_D
  E: All Events depending on I_E
  M: All Modes depending on I_M
begin
  /* Initialisation */
  D := I_D;
  E := I_E;
  M := I_M;
  /* Fixpoint iteration */
  repeat
    /* Transitions that affect interesting data elements
       or have interesting triggers */
    for all m --e,g,f--> m' ∈ Trn with ∃d ∈ D : f updates d
            or ∃d ∈ D : d inactive in m but active in m'
            or e ∈ E do
      M := M ∪{m};
    /* Transitions from or to interesting modes */
    for all m --e,g,f--> m' ∈ Trn with m ∈ M or m' ∈ M do
      D := D ∪ {d ∈ Dat | g reads d }
             ∪ {d ∈ Dat | f updates some d' ∈ D reading d};
      E := E ∪ {e};
      M := M ∪ {m};
    /* Data Flows to interesting data port */
    for all a ⇝ d ∈ Flw with d ∈ D do
      D := D ∪ {d' ∈ Dat | a reads d'};
      M := M ∪ {m ∈ Mod | d := a active in m};
    /* Connections involving interesting event ports */
    for all e ⇝ e' ∈ Con with e' ∈ E do
      E := E ∪ {e, e'};
      M := M ∪ {m ∈ Mod | e ⇝ e' active in m};
  until nothing changes;
end
```

a transitive non-interference definition introduced and explained in [23] and [14]. The theorem they introduced is necessary to connect the (in)dependence that is analysed using program slicing with non-interference. However, a simple extension of this theorem for MILS-AADL specifications that are sliced with the algorithm presented in Section 4.2 is not straight forward, because the program dependency graphs used by Hammer and Snelting differ from the MILS-AADL specifications analysed in this thesis. The main difference is the slicing of automata in the system description, leading to problems concerning non-determinism and different kinds of dependencies (data-dependencies, event-dependencies and mode-dependencies). Nevertheless, it is possible to introduce a similar theorem, allowing to connect non-interference with the slicing technique introduced in Section 4.2, by adapting the non-interference definition such that it is usable for MILS-AADL specifications and different sets containing the data elements, events and modes influencing the slicing criterion.

As the principal idea of slicing is to remove all parts of the program that do not influence the behaviour of interest [18], a sliced system specification only contains parts of a program that influence the result. In case of Goguen-Meseguer non-interference this means that all data elements, events and modes in the backward slice of an observable element must not have a higher security level, because their behaviour has an influence on an observable output. For the transitive Goguen-Meseguer non-interference definition used in [11] this means that an influence between the security domain of the observable element and all elements in its backward slice must be related in a non-interference relation. In the standard interpretation of Goguen-Meseguer non-interference it is assumed that there is an output function describing the reaction of the system depending on the input and that the system is describable by a deterministic state machine. As a result, the Goguen-Meseguer non-interference criterion is restricted to systems, where an input sequence is connected with a single output sequence. Since the automata used to describe the behaviour of a component can be non-deterministic, not every output is connected with specific inputs that will definitely influence the output behaviour. However, it is possible to solve this problem by considering all elements that possibly influence a specific output of the system instead. The inputs possibly interfering with a specific output are given by the inputs determining, which transitions are possibly taken, or by the input values used for computations leading to the output. As these inputs are influencing the output of the system, the sliced system description gained by using Algorithm 4.1 using the output port as input contains all elements that are possibly influencing the output. In other words, we can say that Algorithm 4.1 searches all data elements, events and modes on a trace τ influencing the slicing criterion. Therefore, the elements in the backward-slice of an observable element are describing all elements possibly influencing the output of the system. To express this mathematically, we introduce the relation $I(x, y)$, which expresses that element x possibly has an influence on element y and the function $\sigma(x)$ describing the security level of the element x. This security level is a point in a lattice as described in Chapter 2.2 and further discussed in the next section of this chapter. With this notation it is possible to introduce non-interference as follows.

Definition 4.3.1 (Non-Interference)**.** *Let S be a system description and a be a data or event output port in S. If all elements x possibly influencing a, denoted by $I(x, a)$ have a lower or equal security level $\sigma(x) \sqsubseteq \sigma(a)$, we say that the non-interference criterion is satisfied for a.*

In another way, we can say that a non-interference property for an element a is fulfilled, if all elements having an influence on a (or interfere with a) have a lower or equal security level. According to this non-interference notion, a system is called *secure*, if the non-interference criterion is satisfied for all output ports. Principally, this definition of the non-interference criterion is quite similar to the definition used in [11] and [9] but instead of only allowing deterministic system descriptions here, non-deterministic system descriptions are also allowed by considering sets of possible outputs instead of single outputs.

With this definition of non-interference it is possible to draw a connection between a MILS-AADL system description and non-interference by providing the following theorem:

Theorem 4.3.1. *Let a be a data element, event or mode and d be a data element of a MILS-AADL system specification S. Furthermore, let e be an event in the same specification and m be a mode in S. Additionally, let D, E, M be the result of the slicing algorithm 4.1 with using $\{a\}$ as slicing criterion with the same type (data element, event or mode).*

If

$$d \in D \Rightarrow \sigma(d) \sqsubseteq \sigma(a) \tag{4.1}$$

and

$$e \in E \Rightarrow \sigma(e) \sqsubseteq \sigma(a) \tag{4.2}$$

and

$$m \in M \Rightarrow \sigma(m) \sqsubseteq \sigma(a) \tag{4.3}$$

then the non-interference criterion is satisfied for a.

Proof. We assume that Slicing Algorithm 4.1 is correct. Hence, $d \in D$, $e \in E$ and $m \in M$, we can conclude $I(d, a)$, $I(e, a)$ and $I(m, a)$ by definition of Slicing Algorithm 4.1. Since, $\sigma(d) \sqsubseteq \sigma(a)$, $\sigma(e) \sqsubseteq \sigma(a)$ and $\sigma(m) \sqsubseteq \sigma(a)$ we can follow that the non-interference criterion is satisfied for a. □

This theorem describes how a MILS-AADL specifications can be checked for non-interference using slicing. According to the definition of the backward slice in Section 4.2 all d, e, m influence each other. To be non-interfering the security domain of all these elements influencing the value of a must be allowed to interfere, which means that the relations $\sigma(d) \sqsubseteq \sigma(a)$, $\sigma(e) \sqsubseteq \sigma(a)$ and $\sigma(m) \sqsubseteq \sigma(a)$ must be fulfilled. If this is the case, the system requirement fulfils the non-interference condition, in any other case it is violated and the system must be declared as insecure.

4.4. Security Levels

To combine the information flow-policy and the system description given by a MILS-AADL specification, a method is needed to specify and propagate security levels for data elements, ports and modes. By allowing specification and propagation of security levels, a system developer is able to model the necessary and security critical security levels as initial assignment manually, while the remaining rules are propagated automatically depending on the initial specification in the information flow control. This reduces the specification overhead on the one hand, while maintaining a dynamic security level generation due to propagation on the other hand. The security levels

are considered to be elements of a given lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ describing the information flow-policy.

Hammer and Snelting suggest in [11] to use a function σ to describe actual security levels, a function π to describe the provided security level and the function ρ to describe the required security level of a statement. The provided security level defined by π serves to specify the minimal security level of data provided by this statement, whereas the required security level ρ describes the maximal allowed security level of incoming information. A sufficient definition of an initial security level can be done by only annotating input ports with provided security levels and output ports with required security levels. This definition is sufficient, because a function σ that provides the actual security level is designed to be computed from the required and provided security level. In addition to checking for information leaks, it could also be possible to only specify security levels for input ports in order to calculate the minimal security level needed at the output ports. As this thesis focusses on confidentiality checking, this option is not discussed any further.

As in this thesis MILS-AADL specifications are considered instead of Java bytecode as in [11], not all three functions are necessary for every syntactic structure. Therefore, only the notion of actual security level σ, required security level ρ and provided security level π is adopted, whereas the actual function and propagation rule definition is changed depending on the considered syntactic structure. As a result, the representation in this thesis is changed such that effects only have an actual security level, while ports, variables and modes also have required and provided security levels. A more detailed overview is given in Section 4.4.1.

For a given system description S and a lattice $\mathcal{L}$ a correct information flow control must allow to specify security levels for important syntactic structures and to propagate them along the system. In this process it must be ensured that for confidentiality any data element, event or mode y, influenced by a different data element, event or mode x, must have a greater or equal security level to those elements y. To express that x possibly has an influence on y, the influence relation $I(x, y)$ defined as $I \subseteq (P \cup V \cup M) \times (P \cup V \cup M)$ is introduced. Using this relation the confidentiality requirement from Definition 4.3.1 can be formalised as

$$I(x, y) \Rightarrow \sigma(x) \sqsubseteq \sigma(y) \tag{4.4}$$

In addition to this definition using the influence relation, it can also be claimed that for confidentiality the security level of an element y must be greater or equal to the least upper bound of all elements x influencing y, defined as set $pred(y) = \{x | I(x, y)\}$.

$$\sigma(y) \sqsupseteq \bigsqcup_{x \in pred(y)} \sigma(x) \tag{4.5}$$

A different task for information flow control is integrity. While confidentiality demands that secret information cannot be leaked to public outputs, integrity demands that computations using secret information cannot be manipulated from public input ports. Since integrity can be seen as dual to confidentiality [4], the focus is on confidentiality in this chapter, while integrity is only discussed by extending the constraint (4.5) such that an integrity condition can be expressed. Using the duality to integrity this can be done by changing $\sqsupseteq$ to $\sqsubseteq$, because now y should not be modified by lower classified elements x and $\sqcup$ to $\sqcap$, because the greatest lower bound of all x expresses

the minimal security level of the elements possibly influencing y.

$$\sigma(y) \sqsubseteq \bigsqcap_{x \in pred(y)} \sigma(x) \tag{4.6}$$

To formalize confidentiality it is necessary to prevent security leakages from higher classified inputs to lower classified outputs. This can be done by claiming that no actual security level σ of an expression x should exceed the required security level ρ of x, which is expressed by:

$$\forall x \in dom(\rho) : \rho(x) \sqsupseteq \sigma(x) \tag{4.7}$$

4.4.1. Equations for Security Levels

To specify the needed security levels for important syntactic structures, a method is introduced to calculate these starting from an initial specification. Different syntactic structures have different conditions for specifying confidentiality constraints. Additionally, the derivation of actual security levels from the given provided and required security levels depends on the considered structures. Consequently, it is necessary to differentiate between various rules for ports, variables and effects. However, for the sliced parts only the data elements, which are either data ports or variables, events and modes are interesting. Therefore, the rules for ports, variables and effects are introduced first and recombined to form constraints for data elements, events and modes in a second step. Again, the security levels are considered to be elements of a given lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ describing the flow-policy.

- *Ports:* For simplicity, data ports and event ports are treated in the same way for specifying security levels. This is possible, because the propagation of secret events and ports are logically equivalent. However, as it comes to the analysis the events and data ports are considered separately by either referring to data elements or to events. The same applies for the description of an input or output behaviour. As already mentioned in the introduction of this section, it is assumed that input ports are indicated by having a provided security level π, whereas output ports are having a required security level ρ. To differentiate between provided or required security levels that are introduced by the system developer and ports without a specified security level, partial functions describing initial provided or required security levels and total functions used to describe the security level of all ports used for propagation are introduced. The initial security class of a port is described by the partial functions $\pi'_p : P \nrightarrow L$ and $\rho'_p : P \nrightarrow L$. This assignment maps ports from the set of all system ports P to a lattice point from the domain L, which describes its security level. The function π'_p is used to specify the security level of an input port, whereas the function ρ'_p describes the security level of an output port. As simplification, these functions are extended to total functions $\pi_p : P \to L$ and $\rho_p : P \to L$. In this process all input ports without an initial security level are considered to have the lowest security level possible in $\mathcal{L}$.

$$\pi_p(x) = \begin{cases} \pi'_p(x) & \text{if } x \in dom(\pi'_p) \\ \bot, & \text{else} \end{cases} \tag{4.8}$$

Additionally, unspecified output ports are considered to have the maximal security level in $\mathcal{L}$.

$$\rho_p(x) = \begin{cases} \rho'_p(x), & \text{if } x \in dom(\rho'_p) \\ \top, & \text{else} \end{cases} \tag{4.9}$$

The actual security level $\sigma_p(x)$ of a port x must not only be greater or equal to its own provided security level, but also greater or equal to the security level of ports providing information to x, if x is defined as input port. To express the ports providing information to x, we introduce the function $pred(x)$, whose result contains all output ports that have specified connections or information flows to x. The actual security level $\sigma_p(x)$ can now be expressed as

$$\sigma_p(x) \sqsupseteq \pi_p(x) \sqcup \bigsqcup_{y \in pred(x)} \sigma_p(y) \tag{4.10}$$

In order to prevent leaks it is necessary to introduce the following security constraint claiming that the security level of a port must not exceed its required security level:

$$\rho_p(x) \sqsupseteq \sigma_p(x) \tag{4.11}$$

- *Variables:* The provided security level of a variable is defined in the same way for variables as it is done for ports. Initially, the partial function $\pi'_v : Var \nrightarrow L$ describes an initial assignment of lattice points to variables. The main difference is in the specification of Var, which describes a set of variables v combined with their parent system name s by writing $s.v$. This definition is necessary, to prevent that a variable which is unconnected to a variable with the same name in a different system component is able to effect the security level of the variable in the different system.

$$\pi_v(x) = \begin{cases} \pi'_v(x), & \text{if } x \in dom(\pi'_v) \\ \bot, & \text{else} \end{cases} \tag{4.12}$$

By definition of the MILS-AADL subset variables are unable to leak information to other components without using data ports. Thus, the required security level of a variable is always the maximal security level $\top$ of the lattice $\mathcal{L}$.

$$\rho_v(x) = \top \tag{4.13}$$

The actual security level of a variable x must now be the least upper bound of all variables that can be assigned to x, which is expressed by the set $assign(x)$ containing all variables y that occur on the right hand side of assignments to x.

$$\sigma_v(x) \sqsupseteq \pi_v(x) \sqcup \bigsqcup_{y \in assign(x)} \sigma_v(y) \tag{4.14}$$

- *Effects:* Effects are assignments of the form $x := e$. Even though the consideration of effects is not needed in combination with slicing, a security level definition is introduced to be complete at this point and explain some aspects concerning flow-sensitivity. For this definition, the function $fv(e)$ is introduced,

which contains the variables used in an expression e. Additionally, the function $fp(e)$ describes the set of all ports used in e. The actual security level of an effect $x := e$ is defined as follows.

$$\sigma_{ef}(x := e) = \pi_p(x) \sqcup \bigsqcup_{y \in fv(e)} \sigma_v(y) \sqcup \bigsqcup_{p \in fp(e)} \sigma_p(p) \tag{4.15}$$

With this definition, it could be possible to provide two different security level definitions for the variable x. The first one requires that the actual security level of a variable must be greater or equal to the least upper bound of all effects in a component denoted by the set $Ef(x)$:

$$\sigma_v(x) \sqsupseteq \bigsqcup_{x:=e \in Ef} \sigma_{ef}(x := e) \tag{4.16}$$

This definition has the advantage that it is easy to use and that global variables that might be accessed in subcomponents are threatened in a similar way to data ports. However, it has the disadvantage that an unnecessary incrementation of a security level is possible, which is described in the following example.

Example 4.4.1. Let a system description be given and let l be a variable, which is classified as low in the lattice from Example 2.2.1 in this description. Additionally, let h also be a variable with a high security level in the same system description. Then an effect $l := h$ would result in an incrementation of the security level $\sigma_v(l)$ from L to H. Semantically a security leak is not possible, if the value of the variable stays unchanged for every input value. Therefore, assigning a constant to l before every release of l to a public output would make the system secure. However, an analysis based on this security level definition would detect an security leak, because the actual security level of every mode using l would be at least H but the required security level of the public output port is L.

The second one requires that the security level of a variable is allowed to differ for each effect. Therefore, a dynamic binding method as defined in [12] could be introduced, which allows to define different security levels for the same variable depending on the assigned expression. A different approach would be to define renaming method for variables, which allows a differentiation between two variables containing different results. To avoid the specification overhead that is necessary for these definitions, these two approaches are not described any further and an approach based on slicing is developed in the next sections.

After introducing these basic equations for ports, variables and effects, the security level computation can be extended such that it is applicable for the sets generated by Slicing Algorithm 4.1. The rules introduced for data elements, events and modes are defined by referring to these basic equations as follows.

- *Data Elements:* Data elements are either data ports or variables. By combining these cases it is possible to introduce the function π'_d to describe the initially provided security level of a data element x as follows:

$$\pi'_d(x) = \begin{cases} \pi'_p(x), & \text{if } x \in P \\ \pi'_v(x), & \text{if } x \in V \end{cases} \tag{4.17}$$

Analogously, a total function describing a provided security level for all data elements can be introduced as follows:

$$\pi_d(x) = \begin{cases} \pi_p(x), & \text{if } x \in P \\ \pi_v(x), & \text{if } x \in V \end{cases} \tag{4.18}$$

Likewise, the required security level ρ_d can be introduced as:

$$\rho_d(x) = \begin{cases} \rho_p(x), & \text{if } x \in P \\ \rho_v(x), & \text{if } x \in V \end{cases} \tag{4.19}$$

The actual security level of a data element is the combination of these two cases:

$$\sigma_d(x) \sqsupseteq \pi_d(x) \sqcup \bigsqcup_{y \in allPred(x)} \sigma_d(y) \tag{4.20}$$

where $allPred(x) = pred(x) \cup assign(x)$ describes the union of all port connections of x in $pred(x)$ and all assignments made to the output port in the effects in the considered component $assign(x)$. Finally, the security constraint can be defined as:

$$\rho_d(x) \sqsupseteq \sigma_d(x) \tag{4.21}$$

- *Events:* Every event that occurs is triggered or produced by an event port. Therefore, it is possible to define $\pi'_e(x) = \pi'_p(x)$, $\pi_e(x) = \pi_p(x)$, $\rho_e(x) = \rho_p(x)$ and

$$\sigma_e(x) = \sigma_p(x) \sqsupseteq \pi_p(x) \sqcup \bigsqcup_{y \in pred(x)} \sigma_p(y) \tag{4.22}$$

Finally,

$$\rho_e(x) \sqsupseteq \sigma_e(x) \tag{4.23}$$

is the security constraint for events.

- *Modes:* To cover implicit and explicit information flows the functions $\pi_m : M \to L$ and $\rho_m : M \to L$ are specified for all modes M in the system specification. As for variables it is assumed that the modes are named $c.m$, where c is the component name and m the mode name. The provided security level $\pi_m(x)$ can be derived from the security levels of all effects in outgoing transitions and the events in all incoming transitions. The set of all data elements assigned in effects of outgoing transitions of a mode x is thereby denoted as $O_E(x)$ and $O_{iEvt}(x)$ describes the set of all input event ports used as event triggers of transitions leading to the mode x. Additionally, the set $G(x)$ denotes all data elements that are read by guards of incoming transitions. To get the security level of a transition, the produced events, the data elements read by the guard expression and the data elements that are assigned in the effects have to be considered. Due to the different security level definition for data elements, expressions and modes, the date elements read by guard expressions in incoming transitions and the data elements assigned in outgoing transitions are summarised in the set $D'(x) = \{d | y := d \in O_E(x) \vee g \text{ reads } d \text{ with } g \in G(x)\}$. Additionally, the set $E'(x) = \{e | e \in O_{iEvt}(x)\}$ describes all effects produced at the mode x. The provided security level can now be defined as the least upper bound of the

security levels of all data elements and events that can be forwarded in a mode:

$$\pi_m(x) = \bigsqcup_{d \in D'(x)} \sigma_d(d) \sqcup \bigsqcup_{e \in E'(x)} \sigma_e(e) \tag{4.24}$$

The required security level of a mode depends on the security level of all y in $y := e \in O_E(x)$ and on the required security level of all event port $O_{oEvt}(x)$ used to emit events in incoming transitions. Since a data leak can only occur, if y is an output data port, the required security level can be set to the greatest lower bound of all ports in the effects of outgoing transitions. To differentiate between the required security levels ports and variables in the effects, the function $\rho_y : P \cup V \to L$ is used, that returns the required security level if the input x is a port and $\top$ else.

$$\rho_y(x) = \begin{cases} \rho_p(x), & \text{if } x \in P \\ \top, & \text{else} \end{cases} \tag{4.25}$$

In addition to the data elements, all produced events have to be considered. This is done by defining the set of all input events ports in outgoing transitions as $O_{oEvt}(x)$ and taking the least upper bound of these events. Again we differentiate between the possibly leaked data elements $D''(x) = \{d | d \in d := e \in O_E(x)\}$ and the events $E''(x) = \{e | e \in O_{oEvt}(x)\}$. The required security level of a mode can now be defined as:

$$\rho_m(x) = \bigsqcap_{d \in D''(x)} \rho_y(d) \sqcap \bigsqcap_{e \in E''(x)} \rho_p(evt) \tag{4.26}$$

To prevent implicit information flows the actual security level is defined as:

$$\sigma_m(x) \sqsupseteq \pi_m(x) \sqcup \bigsqcup_{y \in pred(x)} \sigma_m(y) \tag{4.27}$$

Again, to prevent security leaks, we claim that the required security level must be greater or equal to the actual security level of a mode:

$$\rho_m(x) \sqsupseteq \sigma_m(x) \tag{4.28}$$

Transitions are explicitly left out, because their impact on the security level of the system is already expressed in the equations for modes, events and data elements. Based on these equations for data elements, events and modes the confidentiality definition made in the beginning of this section in equations (4.5) and (4.7) can be adapted as follows.

Definition 4.4.1. *Let a system specification S be given, together with a security $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ as well as required- and provided security levels π_p and ρ_p be given for data ports and π_v and ρ_v for variables. The system maintains confidentiality, if all data elements, events and modes satisfy the equations* (4.20)*,* (4.21)*,* (4.22)*,* (4.23)*,* (4.27) *and* (4.28)*.*

In practical applications, calculating the minimal allowed security level of a port or a mode without knowing the actual security level is often more important than checking whether a given security level is sufficient. In order to calculate this minimal allowed security level it is possible to replace the $\sqsupseteq$ in with an $=$ in the (in)equations

(4.20), (4.22), (4.27). This results in the propagation rules for data elements, events and modes:

$$\sigma_d(x) = \pi_d(x) \sqcup \bigsqcup_{y \in allPred(x)} \sigma_d(y) \tag{4.29}$$

$$\sigma_e(x) = \pi_p(x) \sqcup \bigsqcup_{y \in pred(x)} \sigma_p(y) \tag{4.30}$$

$$\sigma_m(x) = \pi_m(x) \sqcup \bigsqcup_{y \in pred(x)} \sigma_m(y) \tag{4.31}$$

These new equations also fulfil the (in)equations (4.20), (4.22) and (4.27), but have the advantage that they can be used for forward propagation of the provided security levels. Consequently, these propagation rules can be used for calculating minimal security levels for each data element, event and mode. Hence, confidentiality can be checked by first deriving all security levels and then checking, whether the security constraints are satisfied. However, this approach has the weakness that in the worst case many security levels for internal variables must be computed that are possibly neither interesting nor useful for future applications. Therefore, the rules are adapted again in Section 4.4.2 by providing theorems, which allow to calculate the actual security level of only chosen elements using the backward slice. To visualise the concept of a confidentiality checking using propagation and to allow a comparison with the approach developed in the next section, Example 4.4.2 is introduced. In this example a system description is outlined that leaks information using various channels and analysed using propagation rules and security constraints.

Example 4.4.2. To familiarise with this basic concept of security levels we consider the following example. We assume that we have given a system description containing a subsystem `leak`, which is intended to leak every information about all inputs to the outputs. The formal system description can be found in the following listing:

```
system example(
  in: in data int 0     // Secret
  out: data int 0       // Public
  e1: in event          // Secret
  e2: out event         // Public
  m0: initial mode

  system leak(
    in: in data int 0
    out: out data int 0
    e1: in event
    e2: out event
    s: bool             // Secret
    x: int 0
    m0: initial mode
    m1: mode
    m0 -[e1 then x := in]-> m1
    m1 -[e2 when s then out := x]-> m0
    m1 -[when not s then out := x]-> m0;
  )
  flow in -> leak.in
  flow leak.out -> out
  connection (e1, leak.e1)
  connection (leak.e2, e2)
)
```

In this example two kinds of information leaks occur. One of them is the explicit information flow from the secret input `in` to the public output `out` via `x`. The second one is called an implicit information flow that occurs, because the event `e2` is produced, if the secret variable `s` evaluates to true. To describe the security levels we use the lattice from Example 2.2.1 as lattice description. The security level L is thereby used to describe the public security level and the security level H is used to describe the secret security level. As initial assignment we get that according to the commented port specification $\pi'_p(example.in) = H$, $\pi'_p(example.e1) = H$, $\rho'_p(example.out) = L$, $\rho'_p(example.e2) = L$. Additionally, the initial security level of `s` is expressed by $\pi'_v(leak.s) = H$. Note that the variable names are always used together with their component specification as it is recommended for variables. By using the equations (4.29), (4.30) and (4.31), it is possible to calculate the security levels of relevant ports by forward propagation, starting from the input ports. The equations (4.21), (4.23) and (4.28) can then be used to check for security leaks.

Using the equations (4.8), (4.9), (4.18), (4.19) and (4.29), we get the provided, required and actual security level for the input port of the example component *example.in*. Using equation (4.21) we get a condition, that has to be assured to prevent security leaks. In our example this means that we get:

$$\begin{aligned}
&\pi_d(example.in) = \pi_p(example.in) = \pi'_p(example.in) = H\\
&\rho_d(example.in) = \rho_p(example.in) = \top = H\\
&\sigma_d(example.in) = \pi_p(example.in) = H\\
&\rho_d(example.in) = H \sqsupseteq H = \sigma_d(example.in)
\end{aligned}$$

By using the same port equations but also the additional event equations (4.30) and (4.23) we get for the input event port *example.e*1:

$$\begin{aligned}
&\pi_e(example.e1) = \pi_p(example.e1) = \pi'_p(example.e1) = H\\
&\rho_e(example.e1) = \rho_p(example.e1) = \top = H\\
&\sigma_e(example.e1) = \sigma_p(example.e1) = \pi_p(example.e1) = H\\
&\rho_e(example.e1) = \rho_p(example.e1) = H \sqsupseteq H = \sigma_p(example.e1)
\end{aligned}$$

The security levels of the subsystem `leak` can be computed using the event port connections and data flows. We get:

$$\begin{aligned}
&\pi_d(leak.in) = \pi_p(leak.in) = \bot = L\\
&\rho_d(leak.in) = \rho_p(leak.in) = \top = H\\
&\sigma_d(leak.in) = \sigma_p(leak.in) = \pi_p(leak.in) \sqcup \sigma_p(example.in) = H\\
&\rho_d(leak.in) = \rho_p(leak.in) = H \sqsupseteq H = \sigma_p(leak.in)\\
&\pi_e(leak.e1) = \pi_p(leak.e1) = \bot = L\\
&\rho_e(leak.e1) = \rho_p(leak.e1) = \top = H\\
&\sigma_e(leak.e1) = \sigma_p(leak.e1) = \pi_p(leak.e1) \sqcup \sigma_p(example.e1) = H\\
&\rho_e(leak.e1) = \rho_p(leak.e1) = H \sqsupseteq H = \sigma_p(leak.e1)
\end{aligned}$$

And for the variables `s` and x we get:

$$\begin{aligned}
&\pi_d(leak.s) = \pi'_v(leak.s) = H\\
&\rho_d(leak.s) = H\\
&\sigma_d(leak.s) = \pi_d(leak.s) = H\\
&\rho_d(leak.s) = H \sqsupseteq H = \sigma_d(leak.s)\\
&\pi_d(leak.x) = L\\
&\rho_d(leak.x) = H\\
&\sigma_d(leak.x) = \pi_d(leak.x) \sqcup \sigma_d(leak.in) = H\\
&\rho_d(leak.x) = H \sqsupseteq H = \sigma_d(leak.x)
\end{aligned}$$

After the security levels of the input ports and variables of the leak component are calculated, we consider the modes using the equations (4.24), (4.26) and (4.31):

$$\begin{aligned}
&\pi_m(leak.m0) = \sigma_d(leak.in) \sqcup \sigma_d(leak.s)\\
&\rho_m(leak.m0) = \rho_y(leak.x) = \top = H\\
&\sigma_m(leak.m0) = \pi_m(leak.m0) = H\\
&\rho_m(leak.m0) \sqsupseteq \sigma_m(leak.m0)\\
&\pi_m(leak.m1) = \sigma_d(leak.x) \sqcup \sigma_e(e1)\\
&\rho_m(leak.m1) = \rho_d(leak.out) \sqcap \rho_e(leak.e2) = \top = H\\
&\sigma_m(leak.m1) = \sigma_m(leak.m0) \sqcup \pi_m(leak.m1)\\
&\rho_m(leak.m1) \sqsupseteq \sigma_m(leak.m1)
\end{aligned}$$

We conclude:

$$\begin{aligned}
&\pi_m(leak.m0) = H\\
&\rho_m(leak.m0) = H \quad \rho_m(leak.m0) = H \sqsupseteq H = \sigma_m(leak.m0)\\
&\sigma_m(leak.m0) = H
\end{aligned}$$

$$\begin{aligned}
&\pi_m(leak.m1) = H\\
&\rho_m(leak.m1) \quad \rho_m(leak.m1) = H \sqsupseteq H = \sigma_m(leak.m1)\\
&\sigma_m(leak.m1) = H
\end{aligned}$$

For the output ports of the example component we get:

$$\begin{aligned}
&\pi_d(example.out) = \pi_p(example.out) = \bot = L\\
&\rho_d(example.out) = \rho_p(example.out) = \rho'_p(example.out) = L\\
&\sigma_d(example.out) = \sigma_p(example.out)\\
&\qquad = \pi_p(example.out) \sqcup \sigma_p(leak.out) \sqcup \sigma_v(leak.x)\\
&\qquad = L \sqcup H = H\\
&\rho_d(example.out) = \rho_p(example.out) = L \sqsupseteq H = \sigma_p(example.out) \quad \lightning\\
&\pi_e(example.e2) = \pi_p(example.e2) = \bot = L\\
&\rho_e(example.e2) = \rho_p(example.e2) = \rho'_p(example.e2) = L\\
&\sigma_e(example.e2) = \sigma_p(example.e2) = \pi_p(example.e2) \sqcup \sigma_p(leak.e2) = L \sqcup H = H\\
&\rho_e(example.e2) = \rho_p(example.e2) = L \sqsupseteq H = \sigma_p(example.e2) \quad \lightning
\end{aligned}$$

Both equations $\rho_d(example.out) \sqsupseteq \sigma_d(example.out)$ and $\rho_e(example.e2) \sqsupseteq \sigma_e(example.e2)$ are violated, and therefore an information leak is detected. But, if the initial required security level of $example.out$ and $example.e2$ is declared as high, the violation disappears, because the ports are now high output ports.

4.4.2. Confidentiality Checking using Slicing

Until this point the equalities and constraints do not fully depend on the slicing approach. To add this relation, it is necessary to reformulate the computation of the actual security level of a data element or event. Therefore, the elements used for the security level propagation are computed using the backward slicing algorithm presented in Algorithm 4.1. As a result, only the elements influencing a statement are considered in order to compute its security level. This approach is justified by the following theorem, which states that the actual security level of any data element, event or mode is given by the least upper bound of all data elements and events it depends on.

Theorem 4.4.1. *Let a system description S be given and x be a data element, event or mode. Let D, E and M be the result of the slicing algorithm 4.1 using $\{x\}$ as slicing criterion as initial set of data elements, events ore modes while the remaining parts are considered to be the empty set. Then the least fixpoint $\sigma(x)$ for the actual security level is gained by*

$$\sigma(x) = \begin{cases} \bigsqcup_{y \in D} \pi_d(y), & \text{if } x \in P_D \cup V \\ \bigsqcup_{y \in E} \pi_e(y), & \text{if } x \in P_E \\ \bigsqcup_{y \in D} \pi_d(y) \sqcup \bigsqcup_{y \in E} \pi_e(y), & \text{if } x \in M \end{cases} \tag{4.32}$$

Proof. To prove this we follow the inductive concept shown in [11] for program dependency graphs. We assume that D, E and M are the result of the Slicing Algorithm 4.1, and differentiate between the following cases:

1. x is a Data Element: Equation (4.29) implies that $\sigma_d(x) \sqsupseteq \pi_d(x)$ and $\sigma_d(x) \sqsupseteq \sigma_d(y)$ for all $y \in pred(x) \cup assign(x)$. By induction, this implies that for any $I(y, x)$: $\pi_d(y) \sqsubseteq \sigma_d(x)$. Since, $I(y, x)$ implies $y \in D$ by definition of Algorithm 4.1, we get $\sigma_d(x) \sqsupseteq \bigsqcup_{y \in D} \pi_d(y)$, by the definition of a supremum. In the next step, we need to show that (4.32) is a (minimal) solution of (4.29). Therefore, we introduce the set $D(y)$, to denote that the set of data elements D is a result of Algorithm 4.1. Due to the fact that we assume $\sigma(x)$ to be the least fixpoint (the minimal security level in the security lattice that is considered as save) we get that $\sigma_d(x) \sqsubseteq \bigsqcup_{y \in D} \pi_d(y)$ and therefore:

$$\begin{aligned} \sigma_d(x) &\sqsubseteq \bigsqcup_{y \in D} \pi_d(y) \\ &= \pi_d(x) \sqcup \bigsqcup_{y \in pred(x) \cup assign(x)} \bigsqcup_{z \in D(y)} \pi_d(z) \\ &\overset{(4.29)}{=} \pi_d(x) \sqcup \bigsqcup_{y \in allPred(x)} \sigma_d(y) \end{aligned}$$

   Consequently, the equality follows as stated in the theorem.

2. x is an Event: Can be shown analogous to the previous case.

3. x is a Mode: By definition of the slicing algorithm 4.1 D and E contain all data elements and events that have an influence on the mode x. From equation (4.31) follows that $\sigma_m(x) \sqsupseteq \pi_m(x)$ and $\sigma_m(x) \sqsupseteq \sigma_m(y)$ for all $y \in pred(x)$. By induction this implies that for any data element y and event z influencing x the provided security levels of y and z are lower or equal to the actual security level of x, which is denoted by $I(y,x) \wedge I(z,x) : \pi_d(y) \sqsubseteq \sigma_m(x) \wedge \pi_e(z) \sqsubseteq \sigma_m(x)$. By definition of the supremum, $\sigma_m(x) \sqsupseteq \bigsqcup_{y \in D} \pi_d(y) \wedge \sigma_m(x) \sqsupseteq \bigsqcup_{z \in E} \pi_e(z)$. From the definition of $\sqsubseteq$, which states that $a \sqsubseteq b$ implies that $a \sqcup b = b$, we can follow:

$$\sigma_m(x) \sqsupseteq \bigsqcup_{y \in D} \pi_d(y) \text{ iff } \sigma_m(x) \sqcup \bigsqcup_{y \in D} \pi_d(y) = \sigma_m(x)$$
$$\sigma_m(x) \sqsupseteq \bigsqcup_{y \in D} \pi_e(y) \text{ iff } \sigma_m(x) \sqcup \bigsqcup_{z \in E} \pi_e(z) = \sigma_m(x)$$

Using the monotonicity of order-theoretic lattices [3] (which are the lattices considered for the security classes) we get that:

$$\begin{aligned}
&\sigma_m(x) \sqsupseteq \bigsqcup_{y \in D} \pi_d(y) \\
\Leftrightarrow &\sigma_m(x) = \sigma_m(x) \sqcup \bigsqcup_{y \in D} \pi_d(y) \\
\Leftrightarrow &\sigma_m(x) \sqcup \bigsqcup_{z \in E} \pi_e(z) = \sigma_m(x) \sqcup \bigsqcup_{y \in D} \pi_d(y) \sqcup \bigsqcup_{z \in E} \pi_e(z) \\
\Leftrightarrow &\sigma_m(x) \sqsupseteq \bigsqcup_{y \in D} \pi_d(y) \sqcup \bigsqcup_{z \in E} \pi_e(z)
\end{aligned}$$

It is now left to show that (4.32) is a (minimal) solution of (4.31). Since we assume $\sigma(x)$ to be the least fixpoint we get that $\sigma(x) \sqsubseteq \bigsqcup_{y \in D} \pi_d(y) \sqcup \bigsqcup_{y \in E} \pi_e(y)$ and therefore:

$$\begin{aligned}
\sigma(x) &\sqsubseteq \bigsqcup_{y \in D} \pi_d(y) \sqcup \bigsqcup_{y \in E} \pi_e(y) \\
&= \pi_m(x) \sqcup \bigsqcup_{y \in pred(x)} \bigsqcup_{z \in M(x))} \pi_d(z) \sqcup \bigsqcup_{z' \in E} \pi_e(z') \\
&\overset{(4.31)}{=} \pi_m(x) \sqcup \bigsqcup_{y \in pred(x)} \sigma_m(y)
\end{aligned}$$

Consequently, equality as stated in the theorem follows. □

Intuitively, a system is secure if no higher classified element can influence the result of a lower classified output port. In the context of provided and actual security level this can be formulated as the condition that the value of all output ports must not depend on higher classified elements than the output port has itself. This can be formulated by the following theorem.

Theorem 4.4.2. *If*

$$\forall a \in dom(\rho'_p) : \forall d \in D(a) \cap dom(\pi'_d) \wedge \forall e \in E(a) \cap dom(\pi'_p) : \\ \pi'_d(d) \sqsubseteq \rho'_p(a) \wedge \pi'_e(e) \sqsubseteq \rho'_p(a) \tag{4.33}$$

then confidentiality is maintained.

Proof. To prove this theorem we need to show that (4.7) is fulfilled for all output ports a, with a specified required security level, and for all data elements d and events e in the backward slice of a. From the premise it is possible to follow that $\forall d \in D(a) \wedge \forall e \in E(a) : \pi_d(d) \sqsubseteq \rho_p(a) \wedge \pi_e(e) \sqsubseteq \rho_p(a)$, because $\pi_d(d) = \pi'_d(d)$, if $d \in (dom(\pi'_p) \cup \pi'_v))$ and $\pi_e(e) = \pi'_e(e)$, if $e \in dom(\pi'_p)$. In the next step we follow $\bigsqcup_{d \in D(a)} \pi_d(d) \sqsubseteq \rho_p(a) \wedge \bigsqcup_{e \in D(a)} \pi_e(e) \sqsubseteq \rho_p(a)$, thus by theorem (4.4.1) we get that $\sigma(d) \sqsubseteq \rho_p(a) \wedge \sigma(e) \sqsubseteq \rho_p(a)$. Consequently (4.7) is satisfied for all data elements and events. □

In addition to the reduced computational overhead, it is possible to derive an confidentiality checking algorithm based on slicing from Theorem 4.4.2, which is presented in Algorithm 4.2. This algorithm takes a system description S and partial functions ρ'_p, π'_d and π'_e as input. While the system description specifies the system that should be analysed, the partial functions describe the required or provided security levels that are defined for the system initially. The output of this algorithm is a boolean variable indicating whether a security leak occurred and a set of security constrains justifying the solution.

The steps necessary to perform are in principle equivalent to the intuitive approach used in Example 4.4.3. For every output port, which is defined as a port with a specified required security level ρ_p, the backward slide is computed. In the next step it is checked if a security leak occurred by formulating and checking the security constraints and the security constraints are added to the output.

Using Algorithm 4.2, the computation overhead can be reduced drastically, because an expensive unnecessary security level computation via forward propagation can be left out. By calculating the backward slice for every $a \in \rho'_p$, it is possible to calculate all elements that are depending on the output ports with an initially specified security level. Having these elements, we can now check whether any element in the backward slice has a higher initial security level than the required security level of the output port. With these two theorems the effort needed to analyse programs can be drastically reduced. To illustrate this we look again at Example 4.4.2 but now use Theorem 4.4.2 together with Algorithm 4.2.

Example 4.4.3. Assuming the same system description as in Example 4.4.2 we can check confidentiality using the presented algorithm based on Theorem 4.4.2 as follows.

Firstly, we calculate the backward slice for every output port with a specified security level ρ'_p. In this example these port are *example.out* and *example.e*2. For reasons of readability, just the results are presented in this example, while the advanced execution is described in Appendix B.1. For this calculation we use the notation $D(example.out), E(example.out)$ and $M(example.out)$ to denote the output of Algorithm 4.1 with *example.out* as initial singleton set of data elements. The same notation is used for the resulting sets for *example.e*2 as singleton set data element input. Thus, we can follow for the data element *example.out* as the first execution of

Algorithm 4.2 Basic Confidentiality Checking Algorithm

```
procedure checkConfidentiality(S, ρ'_p, π'_p, π'_v)
input:
  S: A system description
  ρ'_p: Required security levels of ports in S
  π'_d: Provided security levels of data elements in S
  π'_e: Provided security levels of effects in S
output:
  L: A boolean variable indicating a security leak
  χ: The generated set of security constraints
begin
  /* Initialisation */
  L := false;
  χ := ∅;
  BS := ∅; // A variable used to store the
           // (D, E, M) Slicing triples
  /* Calculate the Backward Slice */
  for all a ∈ dom(ρ'_p) do
    if a is a data element then
      BS := calculateBackwardSlice(S, {a}, ∅, ∅);
    fi
    if a is an effect then
      BS := calculateBackwardSlice(S, ∅, {a}, ∅);
    fi
    /* Generate Constraints and detect security leaks */
    for all (D, E, M) ∈ BS do
      for all d ∈ D do
        if d ∈ dom(π'_d) then
          χ := χ ∪{"π'_d(d) ⊑ ρ_p(a)"};
          if π'_d(d) ⋢ ρ'_p(a)} then
            L := true;
          fi
        fi
      for all e ∈ E do
        if e ∈ dom(π'_e) then
          χ := χ ∪{"π'_d(d) ⊑ ρ_p(a)"};
          if π'_e(d) ⋢ ρ'_p(a)} then
            L := true;
          fi
        fi
end
```

the outer for all loop:

$$D(example.out) = \{example.out, leak.out, example.s, leak.x, leak.in, example.in\}$$
$$E(example.out) = \{leak.e1, leak.e2, example.e1, example.e2\}$$
$$M(example.out) = \{example.m0, leak.m0, leak.m1\}$$

After calculating the backward slice BS, Algorithm 4.2 uses BS to generate and check the following security constraints based on Theorem 4.4.2:

$$\pi'_d(example.in) \sqsubseteq \rho'_p(example.out)$$
$$\pi'_e(example.e1) \sqsubseteq \rho'_p(example.out)$$

As the example has a second output port $example.e2$, the same steps are performed for this output port. In the first step of the outer for all loop, the backward slice using $\{example.e2\}$ as initial event is computed[1]:

$$D(example.e2) = \{leak.s, leak.x, leak.in, example.in\}$$
$$E(example.e2) = \{example.e2, leak.e2, leak.e1, example.e1\}$$
$$M(example.e2) = \{example.m0, leak.m1, leak.m0\}$$

Afterwards, also for $example.e2$ the security constraints are generated and checked based on Theorem 4.4.2:

$$\pi'_d(example.in) \sqsubseteq \rho'_p(example.e2)$$
$$\pi'_e(example.e1) \sqsubseteq \rho'_p(example.e2)$$

Since the security constraints are violated for the initial definition made in Example 4.4.2 an information leak is detected with a reduced overhead.

[1] An extended computation is also presented in Appendix B.1

5. Analysing Cryptographically-Masked Flows

In the previous approach, the security primitives of MILS-AADL were ignored in order to develop a general security analysis without considering the characteristics of cryptographically-masked flows. In this chapter the focus lies on extending this basic approach such that a cryptographically-masked flows can be regarded. However, the consideration of cryptography is not straight forward, because it breaks the notion of standard non-interference. To visualize this problem we assume that a naive modification of slicing algorithm 4.1 only adds y, x, k to D for an encrypt statement in an effect $y :=$ `encrypt(x,k)`. And that for calculating the security constraint the least upper bound of x and k is taken. This approach would be legitimate using the notion of standard non-interference, because the result of the encryption function would depend on x and k. However, releasing the result to a port that is lower classified than x must not necessarily result in an information leak, because the receiver of the encrypted message would not be able to get the secret x from the messages (in an appropriate time) without knowing the decryption key. To underline this problem, in the following example the information flow of the crypto element of the *cryptocontroller* system from Example 3.1.1 is analysed using this idea.

Example 5.0.1. The main purpose of the crypto element from Example 3.1.1 is obviously to encrypt the input and release it to the output. In the most applications this would only be necessary if the output has a lower security level than the input as itself in order to mask the flow for unauthorised entities.

```
system crypto(
  inpayload: in data int 0
  outpayload: out data enc int encrypt(0,k0)
  k: key pub(mykeys)
  m0: initial mode
  m0 -[then outpayload:=encrypt(inpayload,k)]-> m0
 )
```

Therefore, we assume that *crypto.inpayload* has the provided security level H in the security lattice that is shown in Example 2.2.1 and that *crypto.outpayload* has the required security level L. Additionally, it is assumed that k is a public key, which is modelled by assigning the security level L. These statements can be summarised by the initial equations $\pi'_p(crypto.inpayload) = H$, $\rho'_p(crypto.outpayload) = L$ and $\pi'_v(crypto.k) = L$. Using the Slicing Algorithm 4.1 and the theorems presented in Chapter 4.4.2 we get [1]:

$$D(crypto.outpayload) = \{crypto.outpayload, crypto.inpayload, crypto.k\}$$
$$E(crypto.outpalyoad) = \emptyset$$
$$M(crypto.outpayload) = \{crypto.m0\}$$

[1] An extended version of this computation is provided in Appendix B.2

And the security constraints:

$$\pi'_d(crypto.inpayload) = \pi'_p(crypto.inpayload) \sqsubseteq \rho'_p(crypto.outpayload)$$
$$\pi'_d(crypto.k) = \pi'_v(crypto.k) \sqsubseteq \rho'_p(crypto.outpayload)$$

The security constraints are violated, because:

$$\pi'_d(crypto.inpayload) = H, \text{ but}$$
$$\rho'_p(crypto.outpayload) = L$$

Therefore, the program would be considered as insecure. However, the information flow from a high input port to a low output port might be allowed (in this case even is the purpose of the component), if the secret key is not released and the actual value of the high data element is masked by cryptographic operations that are sufficiently strong.

As described in the previous example, a naive extension of the security level assignment to programs with cryptography leads to many false alarms. However, a save extension to cryptographically-masked flows requires that the cryptographic operations used for masking are sufficiently strong. It is easy to think of a situation where poorly encrypted information could be leaked, if the attacker is able to decrypt the information without knowing the private key. Fortunately, it is already assumed in the specification of the MILS-AADL language that messages encrypted with a receivers public key can only be decrypted by receivers knowing the corresponding private key [6].

As a result, the remaining tasks for preventing illegal flows are to detect illegal releases of keys used for cryptographic operations and to evaluate the security level of encrypted and decrypted information. Solving the first task is comparably easy, because it is possible to use the mechanisms preventing classified variables from being leaked. Principally, a public key k is a data element used for encryption and a corresponding private key k' is a data element used to decrypt messages encrypted with k using the decrypt function. To prevent illegal information flows forwarding encryption keys it is therefore possible to treat keys like data elements and to assign security levels analogous to variables. To model the public and the private key with distribution k_p, it is necessary to claim that the security level of the private key $k_{private}$ should be greater or equal to the security level of the public key k_{public} with the same key pair definition:

$$\pi_d(k_{public}) \sqsubseteq \pi_d(k_{private}) \tag{5.1}$$

As a differentiation of public and private keys by their security class is important, it is mandatory to specify provided security levels for encryption keys.

To model the knowledge of an attacker with access to a data port x with security level l it is assumed that he has access to all private keys with a security level $\sqsubseteq l$. This is necessary, because it is not always possible to deduce the keys an attacker has already access to, if he is not modelled as a component within the system. The calculation of the resulting security levels is a bit more complex and is described in the following sections.

5.1. Declassification

As described in the introduction of this chapter, a main problem of the information flow control is its abstraction level, which is too simplistic in cases where information is intentionally released after cryptographic operations. A common approach to handle cryptography in information flow controls, is the concept of *declassification*. Declassification in general describes a mechanism to *downgrade* or *declassify* sensitive information in order to relax the information flow-policy. As a result, declassification is not only suitable for releasing secret information after encryption, but also for any controlled release of information. In their overview paper on declassification Sabelfeld and Sands deal with the question, what the policies for expressing intentional information releases by programs are [22]. As a result to this question, they define a description using the four dimensions:

- *what*: What information is released.
- *who*: Who releases information.
- *where*: Where in the system is the information released.
- *when*: When information can be released.

In addition to these dimensions of declassification, Sablefeld and Sands define four semantic principles of declassification [22]:

- *semantic consistency*: Security definitions should be invariant under equivalence preserving transformations.
- *conservativity*: A declassification should be a weakening of the non-interference definition. In other words this means that without using any declassification, security should reduce to non-interference.
- *monotonicity of release*: Secure programs should not render insecure, if declassification annotations are used. As a result, the security guarantee is weakened the more data objects are declassified.
- *non-occlusion*: The presence of declassification should not allow masking other information flows than the intended ones.

5.2. Security Levels for Cryptographically-Masked Flows

The concept of declassification allows to introduce a more advanced definition of security levels for cryptographic operations. First of all, a key pair must be defined in order to perform a cryptographic operation. This key pair *keys* consisting of public and private keys k and k' must thereby be defined as global constants on top level according to Table 3.1. Any information d which is encrypted with a public key k in the operation `encrypt(d,k)` results in a message m unreadable for any attacker without access to the corresponding private key k'. As a result, it is possible to assume that the result of the encryption always has the security level of the encryption key k. Using the notation of a provided security level the security level definition can be extended for encryption as follows:

$$\sigma(\texttt{encrypt(d,k)}) = \pi_d(k) \tag{5.2}$$

Additionally, to analyse the security levels in the backward slice we add $[k]$ to the set D of dependent data elements to denote that the result of the encryption is possibly any value possible to express in the output of an encryption with public key k. Therefore, the security level of $[k]$ must be the same than for k

$$\pi_d([k]) = \pi_d(k) \tag{5.3}$$

According to Sabelfeld and Sands [22], this policy corresponds to the *where* dimension, because the encryption operation describes the physical location where information is allowed to be declassified. The declassification itself is expressed by ignoring all additional dependencies including d and adding $[k]$ to the data dependencies D instead. This should serve to express that an encrypted message m in the image of `encrypt(d,k)` is assigned at this point. As a result, $\pi_d([k])$ can be used to express the security level of the output of the encryption operation in the analysis.

In addition to the security level computation for encryption using the public key, it is also possible to derive which data elements C are possibly encrypted with the public keys U. This can be done by maintaining (C, U) pairs of all data elements $d \in C$ that are possibly used as first argument of an encryption function `encrypt(d,k)` and all public keys $k \in U$ that are possibly used as second argument. These pairs do not exactly describe the what dimension defined by Sabelfeld and Sands, but allow to analyse the knowledge a possessor of a key in U has. Therefore, it can be used to recalculate the public key of a description. The actual security level of an encryption operation can be defined as least upper bound of all security levels that possibly used encryption keys might have.

For decryption it is necessary to differentiate between two cases. In the first case the decryption is performed in the subsystem of a complete system description. In this case it is possible to reuse the (C, U) pairs calculated in the encryption step in order to derive the information C' that a component has access to, if it knows the private key k'. In order to identify the corresponding private key k' to a public key k in Chapter 3.1 key pairs were introduced to the MILS-AADL syntax to which public and private keys were connected. To draw a mathematical connection we introduce the function $\kappa(k)$ returning the corresponding key pair of a key k. Consequently, a private key k' can be used to decrypt messages encrypted with a public key k if $\kappa(k) = \kappa(k')$. Using this connection C' can be defined as follows.

$$C' = \bigcup \{C | \exists u \in U \exists k' \in K : \kappa(u) = \kappa(k')\} \tag{5.4}$$

The resulting security level is now the least upper bound of all elements in C'.

In the second case it is assumed, that the decryption is performed in the root system or an incomplete system description. Here, the security level can only be regained by taking the security level of the private key, because the (C, U) pair is unknown. Combining both cases leads to the security definition

$$\sigma(\texttt{decrypt(d,k)}) = \bigsqcup_{d \in C} \pi_d(d) \sqcap \bigsqcup_{k \in K} \pi_d(k) \tag{5.5}$$

where $d \in D$ describes a possible argument that is decrypted and $k \in K$ describes a possibly used private key.

To prevent data elements from being leaked using encryption, it is necessary to introduce additional security constraints. A possible leak could occur, if data elements

are encrypted with a public key k and a corresponding private key k' with a lower security level than the encrypted data has itself. As a result, encrypted data elements that can be decrypted with a lower classified public key must be considered as leaked data elements, because anyone with access to the lower classified private key k' is able to read the information. To prevent these misuses of encryption, it is necessary to introduce that all elements in C' must have a security level which is lower or equal to the greatest lower bound of all private keys that are possibly used for decryption. Mathematically, this can be formulated by the following definition.

Definition 5.2.1. *Let C' be the set of all data elements that are possibly decrypted with the private keys in K. Then the decryption is called secure, if*

$$\forall d \in C' : \sigma_d(d) \sqsubseteq \bigsqcap_{k \in K} \pi_d(k) \tag{5.6}$$

By combining encryption and decryption the security level definition of data elements, namely equations (4.20) and (4.29) can be redefined as follows. To analyse encryption, the security level definition of a data element x is modified such that in addition to the variables and ports in $allPred(x)$, also the private keys are considered. To model decryption, all elements in C' must be considered for all private keys K possible to use as decrypted information. To model the encrypted data elements influencing the currently considered data element x, the set $enc(x) = \{k | \exists m, m' \in M \exists y \in P \cup V : \texttt{m -[x := encrypt(y, k)]-> m'}\}$ is introduced. Since we assumed that the result of a cryptographic operation is always declassified to the security level of k, the set $cryptPred(x) = pred(x) \cup assign(x) \cup enc(x) \cup C'(K)$ can be introduced to express the security levels of all data elements influencing x. By using this set, (4.29) can be updated for cryptographic operations as follows:

$$\sigma_d(x) = \pi_d(x) \sqcup \bigsqcup_{y \in encPred(x)} \sigma_d(y) \sqcap \bigsqcup_{k \in K} \sigma_d(k) \tag{5.7}$$

Equation (4.20) can be updated analogously by using the $\sqsupseteq$ instead of the equality.

5.3. Slicing for Cryptographically-Masked Flows

After describing the sets needed for the security level definition in the previous section, the aim of this section is to introduce a mechanism for deriving these sets. As for the general security level description, this mechanism should be based on slicing, to maintain the advantages gained in the previous approach. Therefore, the Slicing Algorithm 4.1 is extended such that the set of possibly encrypted data element C, possibly used private key U and public keys K are calculated for each encryption or decryption operation, while the sets D, E and M should stay the same. The input variables of this algorithm are unchanged. Therefore, I_D, I_E and I_M still describe the sets of interesting data elements, events and modes used as slicing criterion. As result, the algorithm returns a sets D, E and M of data elements, events and modes having an influence on the elements used as slicing criterion. Additionally, the modified slicing algorithm returns a set CU containing all elements used as first and second element in an encryption operation. These sets can be used to derive which data elements are restored using a decryption operation and which keys can be possibly used for decryption. As this derivation is not necessarily a fixpoint iteration, this is done in a

different algorithm checking for confidentiality of a system which is introduced in the next section.

Algorithm 5.1 A Slicing Algorithm for Cryptographically-Masked Flows

```
procedure calculateCryptoBackwardSlice(S,I_D,I_E,I_M)
input:
  S: A System description
  I_D: The initially interesting Data Elements
  I_E: The initially interesting Events
  I_M: The initially interesting Modes
output:
  D: All Data Elements depending on I_D
  E: All Events depending on I_E
  M: All Modes depending on I_M
  K: All private keys used
  CU: All (C,U) pairs
begin
  /* Initialisation */
  D := I_D;
  E := I_E;
  M := I_M;
  K := ∅;
  CU := ∅;
  /* Fixpoint iteration */
  repeat
    /* Transitions that affect interesting data elements
       or have interesting triggers */
    for all m --e,g,f--> m' ∈ Trn with ∃d ∈ D : f updates d
            or ∃d ∈ D : d inactive in m but active in m'
            or e ∈ E do
      M := M ∪{m};
    /* Transitions from or to interesting modes */
    for all m --e,g,f--> m' ∈ Trn with m ∈ M or m' ∈ M do
        if f updates some d' ∈ D reading d and encrypts
             it with public key k then
          E := E ∪ {e};
          M := M ∪ {m};
          C := C ∪ {d};
          U := U ∪ {k};
          CU := CU ∪ (C,U);
          D := D ∪ {[k]};
        else
          if f uses decryption with private key k' then
            D := D ∪ {d ∈ Dat | g reads d};
            E := E ∪ {e};
            M := M ∪ {m};
            K := K ∪ {k'};
          else
```

```
            D := D ∪{d ∈ Dat | g reads d}
                   ∪{d ∈ Dat | f updates some d' ∈ D reading d};
            E := E  ∪ {e};
            M := M  ∪ {m};
          fi
        fi
    /* Data Flows to interesting data ports */
    for all a ⇝ d ∈ Flw with d ∈ D do
      D := D  ∪ {d' ∈ Dat | a reads d'};
      M := M  ∪ {m ∈ Mod | d := a active in m};
    /* Connections involving interesting event ports */
    for all e ⇝ e' ∈ Con with e' ∈ E do
      E := E  ∪ {e, e'};
      M := M  ∪ {m ∈ Mod | e ⇝ e' active in m};
  until nothing changes;
end
```

5.4. Confidentiality Checking for Cryptographically-Masked Flows

The extended Slicing Algorithm 5.1 and the basic confidentiality checking method presented in Algorithm 4.2 allow to introduce an algorithm for confidentiality checking of programs using cryptographic operations. As in Algorithm 4.2 this modified Algorithm 5.2 takes a system description S and functions ρ'_p, π'_d and π'_e describing the initially provided or required security levels of output ports, data elements and events in S. The output of the algorithm is a boolean variable L indicating whether a security leak occurred or not and a set of security constraints χ justifying the result. Differently from Algorithm 4.2, Algorithm 5.1 is used as slicing algorithm and recreates the decrypted values before checking for confidentiality. Therefore, all possibly encrypted data elements C' are computed as described in Section 5.2 for every decryption operation in the backward slice of an output and added to the data dependencies. To restore all dependencies influencing the decrypted values Algorithm 5.1 is used again with $I_D = C'$, $I_E = \emptyset$ and $I_E = M$ as input and the output is added to the slicing results before cryptography. If C' is empty, then a random variable having the security level is added as data dependency for each private key in K to fulfil equation (5.7). As public keys should have a lower or equal security level than private keys according to equation (5.1), this equation is checked and added to the set χ of security constraints. Additionally, the decrypted elements are checked to be secure according to definition 5.2.1. The remaining security constraints are then checked as for programs without cryptography, however with the additional data elements, events and modes added to the investigated sets.

As the concept of declassification is not sound, it is not possible to introduce a theorem similar to Theorem 4.4.2 for an approach based on declassification. However, it is common to show that the approach follows the principles of declassification that are shown in Section 5.1. Since a detailed proof of these concepts would exceed the limits of this thesis only a few basic points about conservativity and non-occlusion are sketched in the following.

- *conservativity*: Without any use of cryptography Algorithm 5.1 computes the same sliced specifications as Algorithm 4.1, because the added code parts are never executed. Therefore, Algorithm 5.2 and Algorithm 4.2 return the same results, because the sliced specifications are the same and no additional statement considering cryptography is executed.

- *non-occlusion*: Since the results of all cryptographic operations are considered to be data dependencies, all events and modes influencing the control flow of the system should be threatened in the same way as without cryptographically-masked flows. Therefore, only masking of (defined) data elements is possible without producing any additional implicit information flows.

Algorithm 5.2 Confidentiality Checking Algorithm for Crypographically-Masked Flows

```
procedure checkCryptoConfidentiality(S,ρ'_p,π'_p,π'_v)
input:
  S: A system description
  ρ'_p: Required security levels in S
  π'_d: Provided security levels of data elements in S
  π'_e: Provided security levels of effects in S
output:
  L: A boolean variable indicating a security leak
  χ: The generated set of security constraints
begin
  /* Initialisation */
  L := false;
  χ := ∅;
  C' := ∅; // The set of possibly decrypted data elements
  BS := ∅; // A variable used to store the (D,E,M,K,CU)
           // results from Slicing
  for all a ∈ dom(ρ'_p)
    /* Calculate the Backward Slice */
    if a is a data element then
      BS := calculateCryptoBackwardSlice(S,{a},∅,∅);
    fi
    if a is an effect then
      BS := calculateCryptoBackwardSlice(S,∅,{a},∅);
    fi
    /* Restore forwarded private keys */
    K := K ∪ calculateCryptoBackwardSlice(S,K,∅,∅).D;
    /* Add possibly decrypted data elements */
    for all (D,E,M,K,CU) ∈ BS do
      /* Calculate C' and add decrypted variables */
      for all (C,U) ∈ CU such that do
        /* Restore forwarded public keys */
        U := U ∪ calculateCryptoBackwardSlice(S,U,∅,∅).D;
        if ∃u ∈ U' ∃k ∈ K' : κ(u) = κ(k) then
          C' := C' ∪ C;
        fi
```

```
    if C' ≠ ∅ then
      D := D ∪ calculateCryptoBackwardSlice(S,C',∅,∅).D;
      E := E ∪ calculateCryptoBackwardSlice(S,C',∅,∅).E;
      M := M ∪ calculateCryptoBackwardSlice(S,C',∅,∅).M;
      K := K ∪ calculateCryptoBackwardSlice(S,C',∅,∅).K;
      CU := CU ∪ calculateCryptoBackwardSlice(S,C',∅,∅).CU;
    else
      D := D ∪ {[k]|k ∈ K};
    fi
    /* Generate security constraints (cryptography) */
    for all (C,U) ∈ CU do
      for all k ∈ K do
        /* Constraints according to (5.1) */
        for all u ∈ U do
          if κ(k) = κ(u) ∈ K then
            χ := χ ∪{"π'_d(u) ⊑ π'_d(k)"};
            if π'_d(u) ⋢ π'_d(k) then
              L := true;
            fi
          fi
        /* Constraints according to Definition 5.2.1 */
        for all c ∈ C do
          χ := χ ∪{"π'_d(c) ⊑ π'_d(k)"};
            if π'_d(c) ⋢ π'_d(k) then
              L := true;
            fi
    /* Generate security constraints (information flows) */
    for all d ∈ D do
      if d ∈ dom(π'_d) then
        χ := χ ∪{"π'_d(d) ⊑ ρ'_p(a)"};
        if π'_d(d) ⋢ ρ'_p(a)} then
          L := true;
        fi
      fi
    for all e ∈ E do
      if e ∈ dom(π'_e) then
        χ := χ ∪{"π'_d(d) ⊑ ρ_p(a)"};
        if π'_e(d) ⋢ ρ_p(a)} then
          L := true;
        fi
      fi
end
```

5.5. Possibilistic Non-Interference

A different approach avoiding the problems gained by introducing declassification could be developed by using possibilisitc non-interference instead of the non-interference definition made in Chapter 4.3. This method is suggested in [2] and has the

advantage that introducing a theorem analogous to Theorem 4.4.2 becomes possible for a changed notion of non-interference. However, we will see that fully verifying a characteristic for possibilisitc non-interference called restrictiveness or hook-up security would exceed the limits of this thesis. Thus, only a small simplification is sketched, which allows a discussion on non-interference and cryptographically-masked flows as in [2] but is not a fully developed proof of all characteristics of possibilisitc non-interference.

The concept of *restrictiveness* or *hook-up security* is presented in [16, 17] and was developed by Daryl McCullough. The main difference to the standard notion of non-interference introduced by Goguen and Meseguer is the ability to handle non-determinism and to describe a composable security property for systems. Since the MILS-AADL system descriptions analysed in this thesis are non-deterministic as well as component based due to their hierarchical structure, this approach is more practicable than the notion standard non-interference. Restrictiveness is a generalisation of non-interference called *possibilistic non-interference*, which describes that a changes of an input port should not influence the possible behaviour of lower classified output ports. To describe this, McCullough introduces the terms *low-level behaviour* and *low-level state*, describing the behaviour observable from with access to low-level outputs and the system parts (states) influencing this behaviour. The principle of *possibilistic non-interference* is now that *high-level* inputs may not change the low-level state of the output. Restrictiveness is originally defined for traces [16] or state machines [17] but can be adapted such that the automaton describing the semantic behaviour of a MILS-AADL component can be considered.

Broadly speaking, an automaton (or better state machine) could be called restrictive as described in [17], if the following statements hold:

- It is input total, which means that no input is blocked, and so every input possibly has effect on the transitions taken in the system description.

- There is an equivalence relation $\equiv_l$ on system states indicating that the states are equivalent on security level l such that:

  1. Inputs may not affect system parts having a lower security level.
  2. The states b, reached from a state a by a transition, should only depend on the inputs triggering the transition and system parts with a security level level lower or equal to the security level of a.
  3. If two states have the same system parts on a level l, then for any possible output sequence leading from one state there must be an equivalent output sequence sequence leading from the other. A system part in this context describes an input or output sequence. In other words, this requirement means that two states with the same input or output behaviour should provide the same output sequence. In addition to this requirement, which only regards the behaviour of the system, also the resulting state must be equivalent in order to produce the same possible system state.

Summed up into two properties for traces starting from the initial state, we can say that there is

- a “write only up” policy prohibiting higher classified inputs from determining lower classified outputs, and

- a “read only down” policy, which means that the possible output sequences of an output port should only be determined by states in a lower or equal equivalence class.

For cryptography, Askarov, Hedin and Sabelfeld argue that the outputs of ideal encryption operations can be seen as l-equivalent, if the output an encryption operation is possibly any value with security level l, which can be assumed if the encrypted message is undistinguishable for the attacker [2]. Arguing for system descriptions in MILS-AADL this means that the possibly used public keys should have a lower or equal security level than the output. This argumentation would render the declassification assumed in Section 5.2 unnecessary, because the results can be seen as k-equivalent for an encryption with public key k and therefore has inherently the same security level then k.

The consideration of non-determinism is more complex and would require a different slicing method which allows to slice for execution traces. Since this approach would definitely exceed the limits of this work, it is left for further research.

6. Case Studies

In this chapter, the results of the previous chapter should be visualised by discussing some case studies. The aim is to analyse a cryptographically-masked communication provided by the cryptographic controller system from [24], which was analysed in different ways in this thesis. The corresponding case study is presented in Section 6.1. After analysing the encryption alone in, the system is extended such that a second component is added in order to decrypt the information the cryptographic controller sends over an insecure channel. This case study is presented in Section 6.2. After explaining the advantages of the approach developed in this thesis are presented, a limitation of this approach is discussed in Section 6.3. Therefore, the cryptographic controller is extended such that the information is provided in frames which are divided into header and payload by a split component.

6.1. Crypto Controller

In this example, the *cryptocontroller* system from Example 3.1.1 is analysed again. In contrast to the analysis presented in the introduction of Chapter 5, in this section the full system is analysed considering the cryptographically-masked flows. The security levels of this system are expressed using the information flow-policies presented in Example 2.2.1. To model the initially defined security levels for the inputs and outputs of the *cryptocontroller* system, it is assumed that the input frame of the cryptographic controller gets a header with security level L and a payload with security level H. The return value is an output frame `outframe` consisting of a header with security level L and an encrypted payload, which should also have security level L. To express this using the provided and required security levels for ports introduced in Chapter 4.4, we use $\pi'_d(cryptocontroller.header) = L$, $\pi'_d(cryptocontroller.payload) = H$, $pi_d(crypto.k) = L$, and $\rho_p(cryptocontroller.outframe) = L$ as initial security level description. The security levels of the subcomponents are thereby left out in order to allow a security level consideration with the least modelling effort as possible. Furthermore, we assume that all names of ports, variables and modes are implicitly renamed such that data elements, events and modes of the form *component.variable* can be used for the analysis. In order to allow a clear distinction between variables with the same name in different (sub-)components, we only allow system descriptions to be analysed, if this clear distinction can be made. Starting from the initial security level assignment, Algorithm 5.2 can be used in order to generate security constraints and to check the confidentiality of the system.

In the first step this algorithm initialises the variables used in the algorithm as follows.

$$L = false, \quad \chi = \emptyset, \quad C' = \emptyset, \quad BS = \emptyset$$

After initialising the variables, confidentiality is checked for each output port of the system. For this purpose, the algorithm calculates the backward slice of this output

using Algorithm 5.1. As this section should give a general overview on confidentiality checking, only the result of this computation is presented here, while the actual compuatation is shown in Appendix B.3.

$$\begin{aligned} D = \{&cryptocontroller.outframe, merge.frame, merge.header, merge.payload,\\ &bypass.outheader, crypto.outpayload, bypass.inheader, [crypto.k],\\ &cryptocontroller.header\}\\ E = \emptyset&\\ M = \{&cryptocontroller.m0, merge.m0, bypass.m0, crypto.m0\}\\ K = \emptyset&\\ CU = \{&(\{crypto.inpayload\}, \{crypto.k\})\} \end{aligned}$$

In the next step Algorithm 5.2 adds all possibly decrypted dependencies to D, E and M and checks for violations of a correct key usage. Since no decryption operation is used in this example and no key is forwarded, there is nothing added in this step. Finally, the security constraints are then calculated and checked leading to the result:

$$\pi'_d([crypto.k]) = \pi'_d(crypto.k) = L \sqsubseteq L = \rho'_p(cryptocontroller.outframe)$$
$$\pi'_d(cryptocontroller.header) = L \sqsubseteq L = \rho'_p(cryptocontroller.outframe)$$

All constraints are satisfied, therefore no leak is detected.

6.2. Secure Communication

The example presented in this section aims to develop a method to signal information over lower classified channels. Every incoming information in the root system is encrypted using a system that is similar to the cryptographic controller system analysed before. In a second component, it is decrypted using a decryption controller system, if an event decrypt is triggered. This has a various field of application possibilities, for example in a automotive engineering or avionics whenever sensor data is send over public channels and has to be decrypted at specific points in time.

The split and merge operations for encryption and decryption are performed analogously to the cryptographic controller system, but with inverted encryption and decrypted results. As the full system definition would unnecessarily exceed this section, the complete system description is presented in Appendix A.2 and only the `decrypto` component used for decryption is presented here.

```
system decrypto(
 inpayload: in data enc int encrypt(0,k0)
 outpayload: out data int 0
 decrypt: in event
 k: key priv(mykeys)
 m0: initial mode
 m0 -[decrypt then outpayload:=decrypt(inpayload,k)]->m0
)
```

To model the initially defined security levels for the inputs and outputs of the root system we use $\pi'_p(seccom.inheader) = L$, $\pi'_p(seccom.inpayload) = H$, $pi'_d(crypto.k) = L$, $\pi'_d(decrypto.k) = H$, $\pi'_p(seccom.decrypt) = H$ and $\rho'_p(seccom.outframe) = H$. Additionally, we assume $\rho'_p(cryptocontroller.outframe) = L$ to model that the data

frames generated by the cryptographic controller are send over an insecure channel. Starting from this assignment, we use Algorithm 5.2 to generate security constraints and to check the confidentiality of the system. Sine Algorithm 5.2 checks confidentiality for each output port with a specified required security level, the following steps would be executed for $cryptocontroller.outframe$ and $seccom.outframe$. However, only the basic steps for slicing and constraint computation of this part should be sketched here, because the resulting backward slice for $cryptocontroller.outframe$ is quite similar to the results we already generated in Section 6.1. In the first step the specification is sliced for $\{cryptocontroller.outframe\}$ leading to:

$$\begin{aligned} D = \{&cryptocontroller.outframe, merge.frame, merge.header, merge.payload,\\ &bypass.outheader, crypto.outpayload, bypass.inheader, [crypto.k],\\ &cryptocontroller.header, seccom.inheader\}\\ E = \emptyset&\\ M = \{&cryptocontroller.m0, merge.m0, bypass.m0, crypto.m0\}\\ K = \emptyset&\\ CU = \{&(\{crypto.inpayload\}, \{crypto.k\})\} \end{aligned}$$

The security constraint generation in the next step leads to:

$$\begin{aligned} \pi'_d([crypto.k]) = \pi'_d(crypto.k) = L \sqsubseteq L = \rho'_p(cryptocontroller.outframe)\\ \pi'_d(seccom.inheader) = L \sqsubseteq L = \rho'_p(cryptocontroller.outframe) \end{aligned}$$

After the security constraint generation for the first element in $dom(\rho'_p)$ is finished, the same steps are executed for the second element in $dom(\rho'_p)$ which is $seccom.outframe$. Firstly, the variables are reset to allow an independent consideration.

$$L = false, \quad \chi = \emptyset, \quad C' = \emptyset, \quad BS = \emptyset$$

Then the backward slice for $seccom.outframe$ is computed using Algorithm 5.1. As in the previous case study, the advanced computation is not shown here but can be found in Appendix B.4. To additionally increase the readability, only the relevant data elements $D' = D \cap (dom(\pi'_p) \cup dom(\rho'_p))$ are shown here, while the complete set is also part of Appendix B.4.

$$\begin{aligned} D' = \{&seccom.outframe, [crypto.k], seccom.inheader\}\\ E = \{&decrypto.decrypt, decryptocontroller.decrypt, seccom.decrypt\}\\ M = \{&seccom.m0, decryptocontroller.m0, merge2.m0, bypass2.m0,\\ &decrypto.m0, split.m0, split.m1, cryptocontroller.m0, merge1.m0\\ &bypass1.m0, crypto.m0\}\\ K = \{&decrypto.k\}\\ CU = \{&(\{crypto.inpayload\}, \{crypto.k\})\} \end{aligned}$$

In the next step Algorithm 5.2 calculates the set C' containing all data elements that are possibly encrypted by the system. As keys can be forwarded, but only the

keys used as second argument are sliced, the possibly forwarded keys influencing the actually used keys must be added. Since no keys are forwarded, there is nothing to add in this step and K and CU stay unchanged. In the next step, the possibly decrypted data element are restored by comparing the possibly used encryption keys in the (C, U) pairs and the possibly used private keys in K. Since *crypto.k* and *decrypto.k* belong to the same key pair *mykeys*, we get the following set C' of decrypted data elements.

$$C' = \{crypto.inpayload\}$$

To add all elements influencing the decrypted values Slicing Algorithm 5.1 is used again to get these dependencies and added to the previously generated sets D, E and M. As before, the slicing itself is again presented in Appendix B.4. The resulting sets are then:

$$\begin{aligned} D' &= \{seccom.outframe, seccom.inheader, seccom.inpayload\} \\ E &= \{decrypto.decrypt, decryptocontroller.decrypt, seccom.decrypt\} \\ M &= \{seccom.m0, decryptocontroller.m0, merge2.m0, bypass2.m0, \\ &\quad decrypto.m0, split.m0, split.m1, cryptocontroller.m0, merge1.m0 \\ &\quad bypass1.m0, crypto.m0\} \\ K &= \{decrypto.k\} \\ CU &= \{(\{crypto.inpayload\}, \{crypto.k\})\} \end{aligned}$$

After restoring the decrypted data elements, Algorithm 5.2 generates and checks security constraints for a correct use of encryption.

$$\begin{aligned} \pi'_d(crypto.k) &\sqsubseteq \pi'_d(decrypto.k) \\ \pi'_d(crypto.inpayload) &\sqsubseteq \pi'_d(decrypto.k) \end{aligned}$$

Since the security level of *crypto.k* is lower than the security level of *decrypto.k* the encryption was legal and no security leak is detected at this point. Additionally, the decrypted element *crypto.inpayload* and the private key *decrypto.k* have equal security levels. Therefore, the decryption was legal and the operation allowed without a security leak.

In the last step, Algorithm 5.2 generates and checks the security constraints for all elements in D and E with a provided security level.

$$\begin{aligned} \pi'_d([crypto.k]) = \pi'_d(crypto.k) = L \sqsubseteq L &= \rho'_p(seccom.outframe) \\ \pi'_d(seccom.inheader) = L \sqsubseteq H &= \rho'_p(seccom.outframe) \\ \pi'_d(seccom.inpayload) = H \sqsubseteq H &= \rho'_p(seccom.outframe) \end{aligned}$$

As non of these security constraints are violated, no security leak is detected as intended in the specification of the system. However, if the security level of *seccom.out frame* would be L a security leak would correctly be detected, because

$$\pi'_d(seccom.inpayload) = H \sqsubseteq L = \rho'_p(seccom.outframe)$$

would be a violation in this case.

6.3. The Limits of this Approach

In this section a problem of the approach developed in this thesis should be visualised in order to show its limits. Therefore, the original system description of the unchanged *cryptocontroller* system from [24] is taken and analysed using the Algorithm 5.2. Since the main problem is in the split component, only this component is shown here while the full specification is provided in Appendix A.3.

```
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]]-> m1
 m1 -[then payload:=frame[1]]-> m0
)
```

To get this problem we assume that the following security level definition for in put and output ports is given.

$$\pi'_d(cryptocontroller.inframt[0]) = L, \quad \pi'_d(cryptocontroller.inframt[1]) = H,$$
$$\rho'_p(cryptocontroller.outframe[0]) = L, \quad \rho'_p(cryptocontroller.outframe[1]) = L,$$

Additionally, we assume that k is a public key with security level L denoted by $\pi'_d(k) = L$. If we calculate the backward slice using Algorithm 5.1, we get the following result.[1]

$$\begin{aligned} D = \{&cryptocontroller.outframe, merge.frame, merge.header, merge.payload,\\ &bypass.outheader, crypto.outpayload, bypass.inheader, [crypto.k],\\ &split.header, split.frame[0], split.frame[1], cryptocontroller.inframe\}\\ E = \emptyset&\\ M = \{&merge.m0, bypass.m0, crypto.m0, split.m0\\ K = \emptyset&\\ CU = \{&(\{crypto.inpayload\}, \{crypto.k\})\} \end{aligned}$$

Resulting from the definition of the split component, the frame $split.frame[1]$ which contains the payload of the information is always part of the sliced system description. This leads to the problem that the complete $inframe$ is added as dependency. Consequently, a security leak is detected, even though there is now possible way of releasing an unencrypted payload. This problem is caused, because the declassification, which allows to model encryption, can only express which information (an encrypted value) is allowed to be released at a specific point of the system description. This problem was already discussed in Chapter 5.5 where a form of restrictiveness is shown which allows to handle this kind of problem. The main difference to the approach developed in this thesis is that the all possible traces for are considered instead of all data elements possibly influencing the output of the system. This would have the advantage that the precision of the analysis would be increased again.

[1] The full computation is presented in Appendix B.5

7. Conclusion

7.1. Summary

In this thesis, a method for analysing cryptographically-masked information flows using slicing was presented. In this context a core subset of the MILS-AADL specification language was introduced in Chapter 3. This core subset extended the system decryption Kevin van der Pol and Thomas Noll used in [24] for their type system such that a distribution of keys can be expressed and a differentiation between data and event ports can be made.

To slice these specifications, a slicing algorithm for AADL specifications developed by Maximilian Odenbrett, Viet Yen Nguyen and Thomas Noll [18] was presented, such that a basic slicing approach became possible. Since non-interference is one of the most frequently used notions in the context of security analysis, a connection between a version of non-interference provided in Definition 4.3.1 and this slicing algorithm was drawn and summarised in Theorem 4.3.1. The main result of this theorem can be concluded by stating that all data elements, events and modes in the backward slice of a system description should be allowed to interfere in the information flow-policy. To express this allowed interference, security levels were introduced and further specified in the next section of this chapter. To minimise the definition overhead, a method was introduced to propagate initial security levels for data or event ports and variables and security constraints were introduce to specify a confidentiality condition based on the function Hammer and Snelting introduced in [11]. Finally, the results were put together in order to algorithmically check confidentiality in Theorem 4.4.2 and Algorithm 4.2.

However, we saw that this approach was unable to handle cryptographically-masked flows, because the declassifying effect of cryptographic operations was not taken into account. Therefore, the previously described algorithms had to be extended such that this consideration became possible. As a result, (C, U) pairs were introduced in order to model the dependence between the encrypted values and the possibly used decryption keys in Chapter 5. Additionally, a set K of possibly used private keys used for decryption was introduced. Using this notation Algorithms 5.1 and 5.2 were developed, which made a consideration of cryptographically-masked flows possible. To show the differences between the declassification of encrypted messages and possibilistic non-interference, this chapter was concluded by a short introduction of restrictiveness.

Based on these Algorithms in Chapter 6 some case studies using a simple cryptographic controller were introduced. In this context an encryption operation was discussed and a secure communication using encryption and decryption was presented. Finally, this chapter was concluded by an example in which the analysis of this thesis gives false alarms and possibilistic non-interference would lead to better results.

7.2. Future Work

As conclusion of this thesis a few directions for future works should be given. Since this work manly focussed on the theoretical aspects and only pseudocode algorithms are given, the practical aspects of this work could be shown in an implementation. This would allow a more refined argumentation about the scalability and precision of the used algorithms. Moreover, an actual implementation would allow to verify security conditions for additional and more advanced case studies as well as real applications.

In addition to the practical aspects, also some theoretic aspects might require further research. As the used non-interference criterion is not fully a possibilistic non-interference condition, some false alarms as the one presented in 6.3 still occur. To solve such problems the slicing algorithm could be extended such that actual computation traces are considered instead of sets containing data elements, events and modes possibly influencing an output port. A possible starting point would be showing restrictiveness as sketched in Chapter 5.5.

Bibliography

[1] T. Amtoft and A. Banerjee. Information flow analysis in logical form. In *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, pages 100–115, 2004.

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

[3] R. Berghammer. *Ordnungen, Verbände und Relationen mit Anwendungen.* Vieweg+Teubner, 2008.

[4] K. J. Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[5] D-Mils Project Consortium. Intermediate languages and semantics transformations for distributed MILS – part 1. Technical Report D3.2, D-Mils Project, Version 1.2, February 2014.

[6] D-Mils Project Consortium. Specification of MILS-AADL. Technical Report D2.1, D-Mils Project, Version 1.3, July 2014.

[7] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[8] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[9] J. A. Goguen and J. Meseguer. *Security policies and security models.* IEEE, 1982.

[10] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Security and Privacy, 1984 IEEE Symposium on*, pages 75–75, April 1984.

[11] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[12] S. Hunt and D. Sands. On flow-sensitive security types. *SIGPLAN Not.*, 41(1):79–90, Jan. 2006.

[13] H. Mantel. Unwinding possibilistic security properties. In *Computer Security-ESORICS 2000*, pages 238–254. Springer, 2000.

[14] H. Mantel, W. Stephan, M. Ullmann, and R. Vogt. Leitfaden für die Erstellung und Prüfung formaler Sicherheitsmodelle im Rahmen von ITSEC und Common Criteria. Technical report, Bundesamt für Sicherheit in der Informationstechnik und Deutsches Forschungszentrum für Künstliche Intelligenz, Version 1.0c, September 2002.

[15] H. Mantel and H. Sudbrock. *Logic-Based Program Synthesis and Transformation: 22nd International Symposium, LOPSTR 2012, Leuven, Belgium, September 18-20, 2012, Revised Selected Papers*, chapter Types vs. PDGs in Information Flow Analysis, pages 106–121. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[16] D. McCullough. Specifications for multi-level security and a hook-up. In *Security and Privacy, 1987 IEEE Symposium on*, pages 161–161, April 1987.

[17] D. McCullough. Noninterference and the composability of security properties. In *Security and Privacy, 1988. Proceedings., 1988 IEEE Symposium on*, pages 177–186, Apr 1988.

[18] M. Odenbrett, V. Nguyen, and T. Noll. Slicing AADL specifications for model checking. pages 217–221. NASA Conference Proceeding CP-2010-216215, 2010.

[19] J. Rushby. *Noninterference, transitivity, and channel-control security policies*. SRI International, Computer Science Laboratory, 1992.

[20] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[21] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher Order Symbol. Comput.*, 14(1):59–91, Mar. 2001.

[22] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5):517–548, Oct. 2009.

[23] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering and Methodology*, 15(4):410–457, Oct. 2006.

[24] K. Van der Pol and T. Noll. Security Type Checking for MILS-AADL Specifications. page 25 Folien. International Workshop on MILS: Architecture and Assurance for Secure Systems, Amsterdam (Netherlands), 20 Jan 2015 - 20 Jan 2015, Jan 2015.

[25] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, Jan. 1996.

A. Additional Examples

A.1. Extended Security-Level Computation

In this Section, a different way to analyse the data flows for the crypto controller example from the introduction of Chapter 5 is presented. In this approach (again) the naive analysis for cryptographically-masked flows without using the backward slice is used. Therefore, a higher overhead is prduced, leading to the same result as in Chapter 5 (the program is falsely considered as insecure). Using the equations from Chapter 4.4 we can calculate the security levels using the propagation rules as follows

$$\pi_d(crypto.inpayload) = \pi'_p(crypto.inpayload) = H$$
$$\rho_d(crypto.inpayload) = H$$
$$\sigma_d(crypto.inpayload) = H$$
$$\pi_d(crypto.k) = \pi'_v(crypto.k) = L$$
$$\rho_d(crypto.k) = H$$
$$\sigma_d(crypto.k) = \pi_d(crypto.k0) = L$$
$$\pi_d(crypto.outpayload) = L$$
$$\rho_d(crypto.outpayload) = \rho'_p(crypto.outpayload) = L$$
$$\sigma_d(crypto.outpayload)$$
$$= \pi_d(crypto.outpayload) \sqcup \sigma_d(crypto.inpayload) \sqcup \sigma_d(crypto.k) = H$$
$$\pi_m(crypto.m0) = \sigma_d(crypto.inpayload) \sqcup \sigma_d(crypto.k) = H$$
$$\rho_m(crypto.m0) = \rho_d(crypto.outpayload) = L$$
$$\sigma(crypto.m0) = \pi_m(crypto.m0) = H$$

and the security constraints

$$\rho_d(crypto.inpayload) = H \sqsupseteq H = \sigma_d(crypto.inpayload)$$
$$\rho_d(crypto.k) = H \sqsupseteq L = \sigma_d(crypto.k)$$
$$\rho_d(crypto.outpayload) = L \sqsupseteq H = \sigma_d(crypto.outpayload)$$
$$\rho_d(crypto.m0) = L \sqsupseteq H = \sigma_d(crypto.m0)$$

A.2. System Description for Secure Communication

```
system seccom(
  inheader: in data int 0
  inpayload: in data int 0
  outframe: out data (int,int) (0,encrypt(0, k0))
  decrypt: in event
```

```
mykeys: key pair
system cryptocontroller(
 header: in data int 0
 payload: in data int 0
 outframe: out data (int,enc int) (0,encrypt(0, k0))
 mykeys: key pair
 m0: initial mode
 system bypass1(
  inheader: in data int 0
  outheader: 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 pub(mykeys)
  m0: initial mode
  m0 -[then outpayload:=encrypt(inpayload,k)]->m0
 )
 system merge1(
  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
 )
 flow header -> bypass1.inheader
 flow payload -> crypto.inpayload
 flow bypass1.outheader -> merge1.header
 flow crypto.outpayload -> merge1.payload
 flow merge.frame -> outframe
)
system decryptocontroller(
 inframe: in data (int,enc int) (0,encrypt(0, k0))
 outframe: out data (int,int) (0,0)
 decrypt: in event
 m0: initial mode
 system split(
  frame: in data (int,enc int) (0,encrypt(0, k0))
  header: out data int 0
  payload: out data int 0
  m0: initial mode
  m0 -[then header:=frame[0]] -> m1
  m1 -[then payload:=frame[1]]-> m0
 )
 system bypass2(
  inheader: in data int 0
  outheader: out data int 0

  m0: initial mode
  m0 -[then outheader:= inheader]-> m0
 )
 system decrypto(
  inpayload: in data enc int encrypt(0,k0)
  outpayload: out data int 0
  decrypt: in event
  k: key priv(mykeys)
  m0: initial mode
  m0 -[decrypt then outpayload:=decrypt(inpayload,k)]->m0
 )
```

```
   system merge2(
    header: in data int 0
    payload: in data int 0
    frame: out data (int,int) (0,0)
    m0: initial mode
    m0 -[then frame:=(header, payload)]-> m0
   )
   flow inframe -> split.frame
   flow split.header -> bypass2.inheader
   flow split.payload -> decrypto.inpayload
   flow bypass2.outheader -> merge2.header
   flow decrypto.outpayload -> merge2.payload
   flow merge1.frame -> outframe
   connection(decrypt, decrypto.decrypt)
  )
  flow inheader -> cryptocontroller.header
  flow inpayload -> cryptocontroller.payload
  flow cryptocontroller.outframe -> decryptocontroller.inframe
  flow decryptocontroller.outframe -> outframe
  connection(decrypt, decryptocontroller.decrypt)
)
```

A.3. Cryptocontroller with Split

```
system cryptocontroller(
 inframe: in data (int,int) (0,0)
 outframe: out data (int,enc int) (0,encrypt(0, k0))
 mykeys: key pair
 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]]-> m1
  m1 -[then payload:=frame[1]]-> m0
 )
 system bypass(
  inheader: in data int 0
  outheader: 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 pub(mykeys)
  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
 )
 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
)
```

B. Computation Tables

In the following more detailed descriptions of the execution of the different slicing algorithms should be provided. Therefore, tables are provided showing how the fixpoint iteration in the slicing algorithms influences the result. For readability reasons, the last step of the fixpoint iteration without a change is omitted.

B.1. Leak Component

Slicing for the output data port `out` leads to the following result.

Iteration	Result
0	$D = \{example.out\}$ $E = \emptyset$ $M = \emptyset$
1	$D = \{example.out, leak.out\}$ $E = \emptyset$ $M = \{example.m0\}$
2	$D = \{example.out, leak.out, leak.s, leak.x, leak.in, example.in\}$ $E = \{leak.e1, leak.e2, example.e1, example.e2\}$ $M = \{example.m0, leak.m1, leak.m0\}$

Slicing for the output event port `e2` leads to the following result.

Iteration	Result
0	$D = \emptyset$ $E = \{example.e2\}$ $M = \emptyset$
1	$D = \emptyset$ $E = \{example.e2, leak.e2\}$ $M = \{example.m0\}$
2	$D = \{leak.s, leak.x, leak.in, example.in\}$ $E = \{example.e2, leak.e2, leak.e1, example.e1\}$ $M = \{example.m0, leak.m1, leak.m0\}$

B.2. Crypto Component

Iteration	Result
0	$D = \{crypto.outpayload\}$ $E = \emptyset$ $M = \emptyset$
1	$D = \{crypto.outpayload, crypto.inpayload, crypto.k\}$ $E = \{crypto.m0\}$ $M = \emptyset$

B.3. Cryptocontroller

Iteration	Result
0	$D = \{cryptocontroller.outframe\}$ $E = \emptyset$ $M = \emptyset$ $K = \emptyset$ $CU = \emptyset$
1	$D = \{cryptocontroller.outframe, merge.frame\}$ $E = \emptyset$ $M = \{cryptocontroller.m0\}$ $K = \emptyset$ $CU = \emptyset$
2	$D = \{cryptocontroller.outframe, merge.frame, merge.header,$ $merge.payload, bypass.outheader, crypto.outpayload\}$ $E = \emptyset$ $M = \{cryptocontroller.m0, merge.m0\}$ $K = \emptyset$ $CU = \emptyset$
3	$D = \{cryptocontroller.outframe, merge.frame, merge.header,$ $merge.payload, bypass.outheader, crypto.outpayload,$ $bypass.inheader, [crypto.k], cryptocontroller.header\}$ $E = \emptyset$ $M = \{cryptocontroller.m0, merge.m0, bypass.m0, crypto.m0\}$ $K = \emptyset$ $CU = \{(\{crypto.inpayload\}, \{crypto.k\})\}$

B.4. Secure Communication

Iteration	Result
0	$D = \{seccom.outframe\}$ $E = \emptyset$ $M = \emptyset$ $K = \emptyset$ $CU = \emptyset$
1	$D = \{seccom.outframe, decryptocontroller.outframe\}$ $E = \emptyset$ $M = \{seccom.m0\}$ $K = \emptyset$ $CU = \emptyset$
2	$D = \{seccom.outframe, decryptocontroller.outframe,$ $merge2.outframe\}$ $E = \emptyset$ $M = \{seccom.m0, decryptocontroller.m0\}$ $K = \emptyset$ $CU = \emptyset$
3	$D = \{seccom.outframe, decryptocontroller.outframe,$ $merge2.frame, merge2.header, merge2.payload,$

	$bypass2.outheader, decrypto.outpayload\}$ $E = \emptyset$ $M = \{seccom.m0, decryptocontroller.m0, merge2.m0\}$ $K = \emptyset$ $CU = \emptyset$
4	$D = \{seccom.outframe, decryptocontroller.outframe,$ $merge2.frame, merge2.header, merge2.payload,$ $bypass2.outheader, decrypto.outpayload, bypass2.inheader,$ $split.header\}$ $E = \{decrypto.decrypt, decryptocontroller.decrypt\}$ $M = \{seccom.m0, decryptocontroller.m0, merge2.m0,$ $bypass2.m0, decrypto.m0\}$ $K = \{decrypto.k\}$ $CU = \emptyset$
5	$D = \{seccom.outframe, decryptocontroller.outframe,$ $merge2.frame, merge2.header, merge2.payload,$ $bypass2.outheader, decrypto.outpayload, bypass2.inheader,$ $split.header, split.frame, cryptocontroller.outframe\}$ $E = \{decrypto.decrypt, decryptocontroller.decrypt,$ $seccom.decrypt\}$ $M = \{seccom.m0, decryptocontroller.m0, merge2.m0,$ $bypass2.m0, decrypto.m0, split.m0, split.m1,$ $cryptocontroller.m0\}$ $K = \{decrypto.k\}$ $CU = \emptyset$
6	$D = \{seccom.outframe, decryptocontroller.outframe,$ $merge2.frame, merge2.header, merge2.payload,$ $bypass2.outheader, decrypto.outpayload, bypass2.inheader,$ $split.header, split.frame, cryptocontroller.outframe,$ $merge.frame\}$ $E = \{decrypto.decrypt, decryptocontroller.decrypt,$ $seccom.decrypt\}$ $M = \{seccom.m0, decryptocontroller.m0, merge2.m0,$ $bypass2.m0,$ $decrypto.m0, split.m0, split.m1, cryptocontroller.m0\}$ $K = \{decrypto.k\}$ $CU = \emptyset$
7	$D = \{seccom.outframe, decryptocontroller.outframe,$ $merge2.frame, merge2.header, merge2.payload,$ $bypass2.outheader, decrypto.outpayload, bypass2.inheader,$ $split.header, split.frame, cryptocontroller.outframe,$ $merge1.frame, merge1.header, merge1.payload,$ $bypass1.outheader, crypto.outheader\}$ $E = \{decrypto.decrypt, decryptocontroller.decrypt,$ $seccom.decrypt\}$ $M = \{seccom.m0, decryptocontroller.m0, merge2.m0,$ $bypass2.m0, decrypto.m0, split.m0, split.m1,$ $cryptocontroller.m0, merge1.m0\}$ $K = \{decrypto.k\}$

	$CU = \emptyset$
8	$D = \{seccom.outframe, decryptocontroller.outframe,$ $merge2.frame, merge2.header, merge2.payload,$ $bypass2.outheader, decrypto.outpayload, bypass2.inheader,$ $split.header, split.frame, cryptocontroller.outframe,$ $merge1.frame, merge1.header, merge1.payload,$ $bypass1.outheader, crypto.outheader, bypass1.inheader,$ $[crypto.k], cryptocontroller.inheader\}$ $E = \{decrypto.decrypt, decryptocontroller.decrypt,$ $seccom.decrypt\}$ $M = \{seccom.m0, decryptocontroller.m0, merge2.m0,$ $bypass2.m0, decrypto.m0, split.m0, split.m1,$ $cryptocontroller.m0, merge1.m0, bypass1.m0, crypto.m0\}$ $K = \{decrypto.k\}$ $CU = \{(\{crypto.inpayload\}, \{crypto.k\})\}$
9	$D = \{seccom.outframe, decryptocontroller.outframe,$ $merge2.frame, merge2.header, merge2.payload,$ $bypass2.outheader, decrypto.outpayload, bypass2.inheader,$ $split.header, split.frame, cryptocontroller.outframe,$ $merge1.frame, merge1.header, merge1.payload,$ $bypass1.outheader, crypto.outheader, bypass1.inheader,$ $s[crypto.k], cryptocontroller.inheader, seccom.inheader\}$ $E = \{decrypto.decrypt, decryptocontroller.decrypt,$ $seccom.decrypt\}$ $M = \{seccom.m0, decryptocontroller.m0, merge2.m0,$ $bypass2.m0, decrypto.m0, split.m0, split.m1,$ $cryptocontroller.m0, merge1.m0, bypass1.m0, crypto.m0\}$ $K = \{decrypto.k\}$ $CU = \{(\{crypto.inpayload\}, \{crypto.k\})\}$

Slicing for the decrypted values `crypto.inpayload` leads to the following result.

Iteration	Result
0	$D = \{crypto.inpayload\}$ $E = \emptyset$ $M = \emptyset$ $K = \emptyset$ $CU = \emptyset$
1	$D = \{crypto.inpayload, cryptocontroller.payload\}$ $E = \emptyset$ $M = \{cryptocontroller.m0\}$ $K = \emptyset$ $CU = \emptyset$
2	$D = \{crypto.inpayload, cryptocontroller.payload$ $seccom.inpayload\}$ $E = \emptyset$ $M = \{cryptocontroller.m0\}$ $K = \emptyset$ $CU = \emptyset$

B.5. Cryptocontroller with Split

Iteration	Result
0	$D = \{cryptocontroller.outframe\}$ $E = \emptyset$ $M = \emptyset$ $K = \emptyset$ $CU = \emptyset$
1	$D = \{cryptocontroller.outframe, merge.frame\}$ $E = \emptyset$ $M = \emptyset$ $K = \emptyset$ $CU = \emptyset$
2	$D = \{cryptocontroller.outframe, merge.frame, merge.header,$ $merge.payload, bypass.outheader, crypto.outpayload\}$ $E = \emptyset$ $M = \{merge.m0\}$ $K = \emptyset$ $CU = \emptyset$
3	$D = \{cryptocontroller.outframe, merge.frame, merge.header,$ $merge.payload, bypass.outheader, crypto.outpayload,$ $bypass.inheader, [crypto.k], split.header\}$ $E = \emptyset$ $M = \{merge.m0, bypass.m0, crypto.m0\}$ $K = \emptyset$ $CU = \{(crypto.inpayload, crypto.k)\}$
4	$D = \{cryptocontroller.outframe, merge.frame, merge.header,$ $merge.payload, bypass.outheader, crypto.outpayload,$ $bypass.inheader, [crypto.k], split.header, split.frame[0],$ $split.frame[1], cryptocontroller.inframe\}$ $E = \emptyset$ $M = \{merge.m0, bypass.m0, crypto.m0, split.m0\}$ $K = \emptyset$ $CU = \{(\{crypto.inpayload\}, \{crypto.k\})\}$