embeddedworld2017
Exhibition&Conference
... it's a smarter world

Analysing Cryptographically-Masked Information Flows in MILS-AADL Specifications

Thomas Noll[†] and Louis Wachtmeister

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

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

In this paper, we propose a formal approach to distinguish between breaking non-interference because of legitimate use of sufficiently strong encryption on the one side, and due to unintended information leaks on the other side. It employs the well-known technique of program slicing to identify (potential) information flows between the data elements of a specification given in a MILS variant of the Architecture Analysis and Design Language (AADL). Moreover, we investigate the relation between our method and an extended notion of non-interference known as possibilistic non-interference, and demonstrate its applicability on a concrete example system.

I. Introduction

Whenever the architecture of a software 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. For this purpose, *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*.

The aim of our work is to develop a formal method for analysing information flow in systems that are specified in a MILS variant of the Architecture Analysis and Design Language (AADL). The underlying semantic notion is that of *non-interference*, which requires that changing secret input values of a system must not affect its publicly observable output. Although this concept is adequate for many applications, it is often violated in the presence of cryptographic operations. In this setting, the main challenge is to distinguish between legitimate "violations" caused by using (sufficiently strong) encryption mechanisms and unintended information leaks that expose information which should be kept secret.

To address this problem, we employ a well-known method for static analysis, program *slicing*. The latter aims to identify (potential) dependencies between program elements, and has manifold applications in debugging, optimisation, maintenance, and model checking of software. Starting with the basic version of a slicing algorithm for MILS-AADL specifications, we develop an extension that handles cryptographically-masked information flows by taking both the "declassifying" effect of encryption operations and the restoration of dependencies by decryption into account. This is accomplished by analysing for each system component which encryption keys are accessible, and which data is (possibly) subject to cryptographic operations.

In order to establish the correctness of our slicing approach, we investigate its relation to an extended notion of non-interference known as *possibilistic non-interference*. The latter allows low outputs to depend on low inputs and, additionally, ciphertexts, but requires that no observation about possible low outputs may reveal information about changes in high inputs, which turns out to provide a natural model for cryptographically-masked information flows. Moreover, we demonstrate the applicability of our method on concrete example systems.

II. The MILS-AADL Language

The specification language MILS-AADL [2], based on the Architecture Analysis and Design Language (AADL) [10], has been developed within the D-MILS project [3] and is intended to serve as the user-facing representation for model-based design of D-MILS systems. Essential features are component definitions in terms of interfaces and implementations, their architecture and interaction through data and event ports, their internal behaviour, and security-related mechanisms such as encryption and authentication.

[†]Supported by the European Community's FP 7 programme under grant agreement no. 318772

Category	Production rules
Type	$\tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{key} \mid \texttt{enc}\ \tau \mid (\tau \ldots, \tau)$
Expression	$e ::= n \mid x \mid e \oplus e \mid (e, \ldots, e) \mid e[n]$
System	$S ::= \texttt{system}\ s(S^*\ P^*\ C^*\ V^*\ M^*\ T^*)$
Port	$P ::= p : \{\texttt{in} \mid \texttt{out}\}\ \{\texttt{event} \mid \texttt{data}\ \tau\}$
Port connection	$C ::= \texttt{connection}\ ([s.]p, [s.]p)$
Variable	$V ::= x : \tau\ [e]$
Mode	$M ::= m : [\texttt{initial}]\ \texttt{mode}$
Transition	$T ::= m -[\,[p]\ [\texttt{when}\ e]\ [\texttt{then}\ \vec{x} := \vec{e}\,]-> m'$

TABLE I
SYNTAX OF MILS-AADL

We analyse models described in a simplified version of MILS-AADL, whose syntax is given in Table I. Here, "{...}" represents a grouping of syntactical elements, "|" stands for alternatives, and "[...]" denote optional elements.

MILS-AADL combines an architectural and a behavioural description of a system s, as follows: one describes a hierarchy of *components*, each possibly containing subsystems, input and output *ports*, and port *connections*. The highest or outermost system in this hierarchy is called the *root* system. This describes the architecture. We distinguish event ports and data ports. *Event* ports can trigger changes in behaviour. *Data* ports are used to communicate data values to or from the environment. The behaviour is defined by a finite automaton with modes and transitions between them. The latter are labelled with an event (port) p (input/output events are consumed/produced when the transition is taken, respectively), a *guard* expression (the transition is enabled only when the guard evaluates to true), and a list of *effects* $\vec{x} := \vec{e}$ (expressions are evaluated in the current state and simultaneously assigned to the left-hand side variables). The event, guard and effects are all optional. If the event is omitted, no port's event will be consumed or produced. If the guard is omitted, it equals true. If the effect is omitted, the system's variables remain unchanged.

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

Example 1: The running example of this paper is taken from [8]. A cryptographic controller is placed between a secure computer and an untrusted network, encrypting all data going from the secure computer to the untrusted network. However, only the payloads of the messages are considered confidential, not their headers. For this reason, the `split` subcomponent is first employed to split up incoming frames into a header ("`hdr`") and a payload ("`load`") part. The former are forwarded without any modification through the `bypass` subcomponent, while the confidentiality of the latter is ensured by encryption in the `crypto` subcomponent. For this purpose, it uses the public part of key `mykey`, which is declared on the top level of the specification. Finally, using the `merge` subcomponent header and (encrypted) payload are re-assembled to outgoing frames which can be sent over an untrusted network. The MILS-AADL code of this system and a visual representation of its architecture are respectively given in Figures 1 and 2. In the latter, solid/dashed lines respectively represent public/confidential information flows.

```
system cryptocontroller(
 inframe: in data (int, int)
 outframe: out data (int, enc int)
 mykey: key
 system split(
  frame: in data (int, int)
  hdr: out data int
  load: out data int
  m0: initial mode
  m0 -[then hdr := frame[0];
            load := frame[1]]-> m0
 )
 system bypass(
  inhdr: in data int
  outhdr: out data int
  m0: initial mode
  m0 -[then outhdr := inhdr]-> m0
 )
 system crypto(
  inload: in data int
  outload: out data (enc int)
  u: public(mykey)
  m0: initial mode
  m0 -[then outload :=
            encrypt(inload, u)]-> m0
 )
 system merge(
  hdr: in data int
  load: in data (enc int)
  frame: out data (int, enc int)
  m0: initial mode
  m0 -[then frame := (hdr, load)]-> m0
 )
 connection (inframe, split.frame)
 connection (split.hdr, bypass.inhdr)
 connection (split.load, crypto.inload)
 connection (bypass.outhdr, merge.hdr)
 connection (crypto.outload, merge.load)
 connection (merge.frame, outframe)
)
```

Fig. 1. MILS-AADL specification of crypto controller

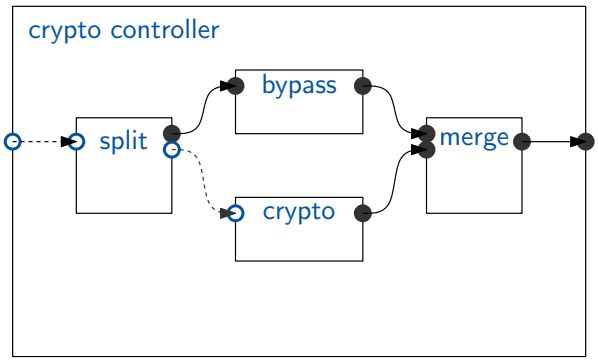

Fig. 2. Architecture of crypto controller

III. Information Flow Security

This section introduces the relevant security concepts. First, we make precise what we mean by security or confidentiality levels. This allows us to further clarify non-interference and to introduce the concept of possibilistic non-interference.

A *security level* describes what data is confidential (secret, private) and what is public. Without loss of generality, we assume that there are two security levels, `H` (high) and `L` (low). Intuitively, the high level means that information is to be kept secret and not be made visible to the outside world. No such restriction is placed on data with a low security level.

Standard *non-interference* [4] requires that low confidentiality outputs may not change when high confidentiality inputs are changed. In a system such as the crypto controller given in Example 1, this is clearly violated as the value of the (low) `outframe` depends on the value of the (high) `inframe`. However, this system has to be considered safe because encryption masks the information from any attackers. Thus, standard non-interference rejects arguably secure and intended uses of encryption.

An extended variant of non-interference aims to repair this defect. In *possibilistic non-interference* [6], [1], we look at the set of possible values after encryption instead of the actual value. We assume that the result of encryption is possibly *any* value in the ciphertext domain. Varying the contents now does not change the possible outcomes of encryption: any ciphertext is still a possible public output. Encryption is an instance of *declassification* [9], where an expression which depends on secret values is not itself secret.

IV. Slicing and Non-Interference

Program *slicing* refers a form of static analysis that can be used to determine (potential) dependencies between the inputs and outputs of system components. Amongst others, this approach has been employed for analysing information flow [5] and for improving the efficiency of model checking [7]. For a given system specification s and a subset $R \subseteq Dat$ of the data elements of s (the so-called *slicing criterion*), the algorithm developed in [7], as shown in Figure 3, identifies

% Slicing of system s with modes Mod, data elements Dat,
% transitions Trn and connections Con w.r.t. criterion R
procedure $slice(R)$:
$D := R; E := \emptyset; M := \emptyset;$
% (1)
repeat
  % Add source modes of transitions
  % that affect interesting data elements
  % or have interesting triggers
  for all $m \xrightarrow{e,g,f} m' \in Trn$ with
      $\exists d \in D : f$ updates d or
      $\exists d \in D : d$ inactive in m but active in m' or
      $e \in E$ **do**
    $M := M \cup \{m\};$
  % Add data elements, events and source modes
  % of transitions from/to interesting modes
  for all $m \xrightarrow{e,g,f} m' \in Trn$ with $m \in M$ or $m' \in M$ **do**
    $D := D \cup \{d \in Dat \mid g$ reads $d\};$
    $E := E \cup \{e\};$
    $M := M \cup \{m\};$
    for all $d' := a$ in f with $d' \in D$ **do**
      $D := D \cup \{d \mid a$ reads $d\};$ % (2)
  % Add sources and modes
  % of connections to interesting data ports
  for all $a \rightsquigarrow d' \in Con$ with $d' \in D$ **do**
    $D := D \cup \{d \in Dat \mid a$ reads $d\};$
    $M := M \cup \{m \in Mod \mid a \rightsquigarrow d'$ active in $m\};$
  % Add sources and targets of connections
  % involving interesting event ports
  for all $e \rightsquigarrow e' \in Con$ with $e \in E$ or $e' \in E$ **do**
    $E := E \cup \{e, e'\};$
    $M := M \cup \{m \in Mod \mid e \rightsquigarrow e'$ active in $m\};$
until nothing changes;
return $(D, E, M);$ % (3)

Fig. 3. Slicing algorithm (labels refer to extensions defined later)

those parts of s which (possibly) have an influence on the slicing criterion by iteratively analysing all data and control dependencies in backward direction. Thus, 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 or indirect influence on the interesting port and thus can be used for a further analysis with regard to 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.

More concretely, it is possible to show [11]: if none of the security levels of the elements of the backward slice of a slicing criterion R exceeds the minimal security level in R, then the system ensures (standard) non-interference.

Example 2: The non-interference property is clearly violated for the crypto controller shown in Figure 1 as the (high) incoming data port `inframe` is in the backward slice of

the (low) outgoing data port `outframe`. This observation is justified by the following chain of dependencies induced by port connections ($\leftarrow_c$) or variable assignments occurring in transitions ($\leftarrow_t$):

$$\begin{aligned}\texttt{outframe} &\leftarrow_c \texttt{merge.frame}\\ &\leftarrow_t \texttt{merge.load}\\ &\leftarrow_c \texttt{crypto.outload}\\ &\leftarrow_t \texttt{crypto.inload}\\ &\leftarrow_c \texttt{split.load}\\ &\leftarrow_t \texttt{split.frame}\\ &\leftarrow_c \texttt{inframe}\end{aligned}$$

V. Handling Cryptographic Operations

As described in Section II, MILS-AADL provides security operations `encrypt` and `decrypt` that respectively require a public and a private key as second argument. These key pairs have to be declared as global constants on the top level of the specification ("`mykey : key`"). Their public/private subkeys can be assigned to variables (such as "`u : public(mykey)`" in `crypto`) and forwarded via data ports. Thus, we are dealing with a *static pool* of keys with *dynamic distribution*. As pointed out earlier, we assume that a message encrypted with a public key can only be decrypted when the corresponding private key is used.

With respect to security analysis, this raises two challenges: we must be able to detect "illegal" releases of keys, and we have to properly evaluate the security level of encrypted and decrypted information. The first problem is already covered by the slicing technique presented in Section IV: if a (high) private key is forwarded to a data port that is only expected to hold a (low) public key, this clearly constitutes a violation of a security policy that can be checked by slicing.

In order to handle the masking effect of cryptographic operations, we employ a *declassification* approach [9]: the information flow policy is relaxed by downgrading encrypted sensitive information, i.e., by assigning a lower security level to the encrypted value. If this value is later decrypted using the matching private key, the result is re-classified to its original security level.

More concretely, our analysis is implemented by an extension of the algorithm shown in Figure 3. It computes a *conditional slice* of the given specification whose contents depends on the distribution of (private) keys in the system. For each usage of an encryption operation of the form $\texttt{encrypt}(d,k)$, we maintain two sets which are both computed by slicing. The first is denoted by C and collects all data elements that possibly influence the message argument d. The second is denoted by U and contains all public keys that may be used as the key argument k. Both are combined to a (C,U) pair. The result of this encryption is always declassified to `L`.

For each usage of a decryption operation of the form $\texttt{decrypt}(d,k)$, the (C,U) pairs that may contribute to the ciphertext argument d are determined using backward slicing, which yields a set CU of such pairs. Moreover, we compute a set P of private keys that may be used as the key argument k by applying slicing again. At this point, it is possible to identify all plaintexts that may result from this decryption operation by taking the union of all C sets whose U parts intersect with P, that is,

$$\bigcup_{(C,U)\in CU} \{C \mid U \cap P \neq \emptyset\}$$

is added to the backward slice of $\texttt{decrypt}(d,k)$. Accordingly, the resulting security level is the maximal level of the data elements in C'.

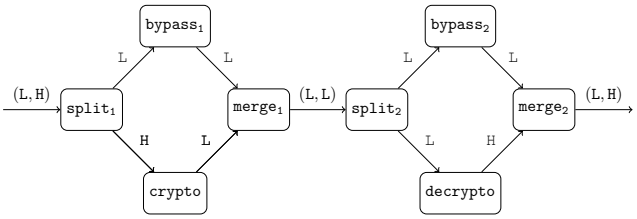

Fig. 4. Combination of encryption/decryption controller

In order to implement this analysis, the slicing algorithm is extended by the following steps that are added at the respective labels given in Figure 3:

1) The set of (C,U) pairs is initialised using the assignment
$$CU := \emptyset.$$
2) The unconditional update $D := D \cup \{d \mid a \text{ reads } d\}$ is replaced by the following case distinction, which collects the (C,U) pairs and private keys in case of an encryption/decryption operation and accordingly extends the slicing information as described before:
   if $a = \texttt{encrypt}(d,k)$ **then**
   $(C, CU', E', M') := \mathit{slice}(\{d\})$;
   $(U, CU'', E'', M'') := \mathit{slice}(\{k\})$;
   $CU := CU \cup \{(C,U)\}$;
   else if $a = \texttt{decrypt}(d,k)$ **then**
   $(D', CU', E', M') := \mathit{slice}(\{d\})$;
   $(P, CU'', E'', M'') := \mathit{slice}(\{k\})$;
   $D := D \cup \{C \mid (C,U) \in CU', U \cap P \neq \emptyset\}$;
   else
   $D := D \cup \{d \mid a \text{ reads } d\}$;
3) After replacing the original **return** statement by
$$\textbf{return } (D, CU, E, M),$$
a call of the extended slicing algorithm additionally yields the (C,U) pairs collected for encryption operations.

In analogy to the correctness result stated at the end of Section IV, one can show that if none of the security levels of the elements of the backward slice of a slicing criterion R exceeds the minimal security level in R, then the system ensures possibilistic non-interference.

Example 3: In order to illustrate the working principle of the conditional slicing algorithm, we extend the crypto controller system from Example 1 by adding a "mirrored"

decryption component `decryptocontroller` as shown in Figure 4. The specification of `decryptocontroller` is almost identical to that of `cryptocontroller` with the exception that the `decrypto` component employs a transition assignment of the form

$$\texttt{outload} := \texttt{decrypt}(\texttt{inload}, \texttt{p})$$

where the private key `p` is declared by $\texttt{p} : \texttt{private}(\texttt{mykey})$ such that $P = \{\texttt{mykey}\}$. Backward slicing of the extended specification with respect to criterion

$$R = \{\texttt{decryptocontroller.outframe}\}$$

first identifies the dependence of `decrypto.outload`, whose value is given by $\texttt{decrypt}(\texttt{decrypto.inload}, \texttt{p})$. A call of $slice(\{\texttt{decrypto.inload}\})$ yields the dependency of `crypto.outload`. An analysis of the defining expression $\texttt{encrypt}(\texttt{crypto.inload}, \texttt{u})$ then returns the singleton pair $CU = \{(C, U)\}$, where C is given by

$$\{\texttt{split}_1\texttt{.load}, \texttt{split}_1\texttt{.frame}, \texttt{cryptocontroller.inframe}\}$$

and $U = \{\texttt{mykey}\}$. Hence, $U \cap P \neq \emptyset$ such that altogether `decryptocontroller.outframe` is dependent on `cryptocontroller.inframe`. However, Figure 4 shows that there is a security assignment that satisfies the requirement of possibilistic non-interference.

VI. Conclusion

In this paper, we have proposed a formal approach to analyse information flow security properties of distributed systems that are specified in a MILS variant of the Architecture Analysis and Design Language (AADL). Due to the presence of encryption features, the standard notion of non-interference is too strong to distinguish between breaking non-interference because of legitimate use of sufficiently strong encryption on the one side, and due to unintended information leaks on the other side. Our key idea is to employ the well-known technique of program slicing to identify (potential) information flows between the data elements of a given specification. Starting with a basic version of a slicing algorithm which does not support encryption (and which essentially characterises non-interference therefore), we developed an extension that keeps track of the distribution and application of keys. This enabled us to clearly identify the (high) inputs on which a (low) output possibly depends in the presence of cryptographic operations. We also investigated the relation between our slicing approach and an extended notion of non-interference known as possibilistic non-interference, and demonstrated its applicability on a concrete example system.

References

[1] A. Askarov, D. Hedin, and A. Sabelfeld, "Cryptographically-masked flows," *Theor. Comput. Sci.*, vol. 402, no. 2-3, pp. 82–101, 2008. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2008.04.028

[2] "Specification of AADL/MILS," D-MILS Project, Tech. Rep. D2.1, Version 1.4, Sep. 2014, http://www.d-mils.org/page/results.

[3] "The D-MILS project web site," http://www.d-mils.org/.

[4] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[5] 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*, vol. 8, no. 6, pp. 399–422, 2009.

[6] D. McCullough, "Noninterference and the composability of security properties," in *1988 IEEE Conference on Security and Privacy*. IEEE Computer Society, 1988, pp. 177–186.

[7] M. Odenbrett, V. Nguyen, and T. Noll, "Slicing AADL specifications for model checking," in *Proc. 2nd NASA Formal Methods Symp. (NFM 2010)*. NASA Conference Proceeding CP-2010-216215, 2010, pp. 217–221. [Online]. Available: http://shemesh.larc.nasa.gov/NFM2010/proceedings/NASA-CP-2010-216215.pdf

[8] J. Rushby, "Noninterference, transitivity, and channel-control security policies," Computer Science Laboratory, SRI International, Tech. Rep. SRI-CSL-92-2, Dec. 1992.

[9] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009. [Online]. Available: http://dx.doi.org/10.3233/JCS-2009-0352

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

[11] L. Wachtmeister, "Analysing cryptographically-masked information flows using slicing," Bachelor Thesis, RWTH Aachen University, Germany, 2016. [Online]. Available: https://moves.rwth-aachen.de/wp-content/uploads/thesisProjects/bsc-thesis-wachtmeister.pdf