Automatic Information Flow Validation for High-Assurance Systems

Kevin Müller*, Sascha Uhrig*, Flemming Nielson†, Hanne Riis Nielson†,
Ximeng Li‡†, Michael Paulitsch§ and Georg Sigl¶
*Airbus Group · Munich, Germany · Email: [Kevin.Mueller|Sascha.Uhrig]@airbus.com
†DTU Compute · Technical University of Denmark · Email: [fnie|hrni|ximl]@dtu.dk
‡Technische Universität Darmstadt · Darmstadt, Germany · Email: li@mais.informatik.tu-darmstadt.de
§Thales Austria GmbH · Vienna, Austria · Email: Michael.Paulitsch@thalesgroup.com
¶Technische Universität München · Munich, Germany · Email: sigl@tum.de

Abstract—**Nowaydays, safety-critical systems in high-assurance domains such as aviation or transportation need to consider secure operation and demonstrate its reliable operation by presenting domain-specific level of evidences. Many tools for automated code analyses and automated testing exist to ensure safe and secure operation; however, ensuring secure information flows is new in the high-assurance domains. The Decentralized Label Model (DLM) allows to partially automate, model and prove correct information flows in applications' source code. Unfortunately, the DLM targets Java applications; hence, it is not applicable for many high-assurance domains with strong real-time guarantees. Reasons are issues with the dynamic character of object-oriented programming or the in general uncertain behaviors of features like garbage collectors of the commonly necessary runtime environments. Hence, many high-assurance systems are still implemented in C. In this article, we discuss DLM in the context of such high-assurance systems. For this, we adjust the DLM to the programming language C and developed a suitable tool checker, called *Cif*. Apart from proving the correctness of information flows statically, Cif is able to illustrate the implemented information flows graphically in a dependency graph. We present this power on generic use cases appearing in almost each program. We further investigate use cases from the high-assurance domains of avionics and railway to identify commonalities regarding security. A common challenge is the development of secure gateways mediating the data transfer between security domains. To demonstrate the benefits of Cif, we applied our method to such a gateway implementation. During the DLM annotation of the use case's C source code, we identified issues in the current DLM policies, in particular, on annotating special data-dependencies. To solve these issues, we extend the data agnostic character of the traditional DLM and present our new concept on the gateway use case. Even though this paper uses examples from aviation and railway, our approach can be applied equally well to any other safety-critical or security-critical system. This paper demonstrates the power of Cif and its capability to graphically illustrate information flows, and discusses its utility on selected C code examples. It also presents extension to the DLM theory to overcome identified shortcomings.**

Index Terms—**Security; High-Assurance; Information Flow; Decentralized Label Model**

I. Introduction

Safety-critical systems in the domains of aviation, transportation systems, automotive, medical applications or industrial control have to show their correct implementation with a domain-dependent level of assurance. Due to the changing IT environments and the increased connectivity demands in the recent years, these system do not operate isolated anymore. Moreover, they are subject of attacks that require additional means to protect the security of the systems. The use cases discussed by this article are derived by the safety and security demands of the avionic and railway domains, both highly restricted and controlled domains for high-assurance systems. This article extends our previous contribution [1] on presenting how security-typed languages can improve the code quality and the automated assurance of correct implementation of C programs, with use cases from both mentioned domains. Furthermore, the paper will provide improvements to the theory of the Decentralized Label Model (DLM); being anon an example for security-typed technologies.

Aviation software [2] and hardware [3] have to follow strict development processes and require certification by national authorities. Recently, developers of avionics, the electronics on-board of aircrafts, have implemented systems following the concepts of Integrated Modular Avionics (IMA) [4] to reduce costs and increase functionality. IMA achieves a system design of safe integration and consolidation of applications with various criticality on one hardware platform. The architecture depends on the provision of separated runtime environments, so called partitions. Targeting security aspects of systems, a similar architectural approach has been developed with the concept of Multiple Independent Levels of Security (MILS) [5]. This architectural approach depends on strict separation of processing resources and information flow control. A *Separation Kernel* [6] is a special certifiable operating system that can provide both mentioned properties.

Apart from having such architectural approaches to handle the emerging safety and security requirements for high assurance systems, the developers also have to prove the correct implementation of their software applications. For safety, the aviation industry applies various forms of code analysis [7][8][9] in order to evidently ensure correct implementation of requirements. For security, in particular on secure information flows, the aviation industry only has limited means available, which are not mandatory yet.

The base for *secure* or *correct information flows* in this paper are security policies for systems that contain rules on flow restrictions from input to outputs of the system, or fine-grained, between variables in a program code. On secure information flow, the DLM [10] is a promising approach.

DLM introduces annotations into the source code. These annotations allow to model an information flow policy directly on source code level, mainly by extending the declaration of variables. This avoids additional translations between model and implementation. Tool support allows to prove the implemented information flows and the defined flow policy regarding consistency. In short, DLM extends the type system of a programming language to assure that a security policy modeled by label annotations of variables is not violated in the program flow.

Our research challenge here is to apply this model to a recurring generic use case of a gateway application. After analyzing use cases of two high assurance industries, we identified this use case as a common assurance challenge in both, the avionic and railway industry. DLM is currently available for the Java programming language [11]. Java is a relatively strongly typed language and, hence, appears at first sight as a very good choice. However, among other aspects the dynamic character of object-oriented languages such as Java introduces additional issues for the certification process [12]. Furthermore, common features such as the Java Runtime Environment introduces potentially unpredictable and harmful delays during execution. For high-criticality applications this is not acceptable as they require high availability and real-time properties like low response times. Hence, as most high-assurance systems remain to be implemented in C, our first task is the adaption of DLM to the C language. Then, we leverage the compositional nature of the MILS architecture to deliver overall security guarantees by combining the evidences of correct information flow provided by the DLM-certified application and by the underlying Separation Kernel. This combination of evidences will also help to obtain security certifications for such complex systems in the future.

In this article we will discuss the following contributions:

DLM for C language: We propose an extension of the C language in order to express and check information flow policies by code annotations; we discuss in Section IV the challenges in adapting to C rather than Java.

Real Use-Case Annotations: While DLM has been successfully developed to deal with typical applications written in Java, we investigate the extent to which embedded applications written in C present other challenges. To be concrete we study in Sections V-VI the application of the DLM to a real-world use case from the avionic and railway domains, namely a demultiplexer that is present in many high security-demanding applications, in particular in the high assurance gateway being developed as a research demonstrator.

Graphical Representation of Information Flows: To make information flow policies useful for engineers working in avionics and automotive, we consider it important to develop a useful graphical representation. To this end we develop a graphical format for presenting the information flows. This helps engineers to identify unspecified flows and to avoid information leakage due to negligent programming.

Improvements to DLM Theory: It turns out that the straight adaptation of DLM to real source code for embedded systems written in C gives rise to some overhead regarding code size increase. In order to reduce this overhead, we suggest in Section IX improvements to the DLM so as to better deal with the content-dependent nature of policies as is typical of systems making use of demultiplexers.

This article is structured as follows: Section II discusses recent research papers fitting to the topic of this paper. In Section III, we introduce the DLM as described by Myers initially. Our adaptation of DLM to the C language and the resulting tool checker *Cif* are described in Section IV. In Section V, we discuss common code snippets and their verification using *Cif*. This also includes the demonstration of the graphical information flow output of our tool. Section VI and Section VII present the security domains inside the aviation and railway industry to motivate our use case. Section VIII discusses this high assurance use case identified as challenging question of both domains. The section further connects security-typed languages with security design principles, such as MILS. In this chapter, we also assess our approach and identify shortcomings in the current DLM theory. Section IX uses the previous assessment and suggests improvements to the DLM theory. Finally, we conclude our work in Section X.

II. Related Work

Sabelfeld and Myers present in [13] an extensive survey on research of security typed languages within the last decades. The content of the entire paper provides a good overview to position the research contribution of our paper.

The DLM on which this paper is based was proposed by Myers and Liskov [10] for secure information flow in the Java language. This model features decentralized trust relation between security principals. Known applications (appearing to be of mostly academic nature) are:

- *Civitas*: a secure voting system
- *JPmail*: an email client with information-flow control
- *Fabric, SIF* and *Swift*: being web applications.

In this paper, we adapt DLM to the C programming language, extending its usage scope to high-assurance embedded systems adopted in real-world industry.

An alternative approach closely related to ours is the Data Flow Logic (DFL) proposed by Greve in [14]. This features a C language extension that augments source code and adds security domains to variables. Furthermore, his approach allows to formulate flow contracts between domains. These annotations describe an information flow policy, which can be analyzed by a DFL prover. DFL has been used to annotate the source code of a Xen-based Separation Kernel [15]. Whereas Greve builds largely on Mandatory Access Control, we base our approach on Decentralized Information Flow Control. The decentralized approach introduces a mutual distrust among data owners, all having an equal security level. Hence, DLM avoids the automatically given hierarchy of the approaches of mandatory access control usually relying on at least one super user.

III. Decentralized Label Model (DLM)

The DLM [10] is a language-based technology allowing to prove correct information flows within a program's source code. Section III-A introduces the fundamentals of the model. The following Section III-B focusses on the information flow control.

A. General Model

The model uses *principals* to express flow policies. By default a mutual distrust is present between all defined principals. Principals can delegate their authority to other principals and, hence, can issue a trust relation. In DLM, principals own data and can define read (confidentiality) and write (integrity) policies for other principals in order to allow access to the data. Consequently, the union of owners and readers or writers respectively defines the effective set of readers or writers of a data item. DLM offers two special principals:

1) Top Principal `*`: As owner representing the set of all principals; as reader or writer representing the empty set of principals, i.e., effectively no other principal except the involved owners of this policy
2) Bottom Principal _: As owner representing the empty set of principals; as reader or writer representing the set of all principals.

Additional information on this are described in [16]. In practice *labels*, which annotate the source code, express the DLM policies. An example is:

```
int {Alice->Bob; Alice<-_} x;
int {*->_; *<-*} y;
```

Listing 1. Declaration of a DLM-annotated Variable

This presents a label definition using curly brackets as token[1]. In this example the principal `Alice` owns the data stored in the integer variable `x` for both the confidentiality and integrity policy. The first part of the label `Alice->Bob` expresses a confidentiality policy, also called reader policy. In this example the owner Alice allows Bob to read the data. The second part of the label expresses an integrity policy, or writer policy. In this example it defines that Alice allows all other principals write access to the variable `x`. For the declaration of `y` the reader policy expresses that all principals believe that all principals can read the data and the writer policy expresses that all principals believe that no principal has modified the data. Overall, this variable has low flow restrictions.

In DLM one may also form a conjunction of principals, like `Alice&Bob->Chuck`. This confidentiality policy is equivalent to `Alice->Chuck;Bob->Chuck` and means that the beliefs of Alice and Bob have to be fulfilled [17].

[1]In the following we will use the compiler technology-based term *token* and the DLM-based term *annotation* as synonyms.

B. Information Flow Control

Using these augmentations on a piece of source code, a static checking tool is able to prove whether all beliefs expressed by labels are fulfilled. A data flow from a source to an *at least equally restricted* destination is a *correct* information flow. In contrast an invalid flow is detected if data flows from a source to a destination that is less restricted than the source. A destination is *at least as restricted* as the source if:

- the confidentiality policy keeps or increases the set of owners and/or keeps or decreases the set of readers, and
- the integrity policy keeps or decreases the set of owners and/or keeps or increases the set of writers

```
int {Alice->Bob; Alice<-_}
  x = 1;
int {Alice&Bob->*; Alice<-_}
  y = 0;

y = x;
```

Listing 2. Valid Direct Information Flow

```
int {Alice->Bob; Alice<-_}
  x = 1;
int {Alice&Bob->*; Alice<-_}
  y = 0;

if(y == 0)
  x = 0;
```

Listing 3. Invalid Implicit Information Flow

Listing 2 shows an example of a valid direct information flow from the source variable x to the destination y. Apart from these direct assignments, DLM is also able to detect invalid implicit flows. The example in Listing 3 causes an influence on variable x if the condition $y == 0$ is true. Hence, depending on the value of y the data in variable x gets modified, i.e., allowing x to observe the status of y. However, y is more restrictive than x, i.e., x is not allowed to observe the value of y. Thus, the flow in Listing 3 is invalid.

To analyse those implicit flows, DLM also examines each instruction against the current label of the Program Counter (PC). As in Java Information Flow (Jif) [18], the PC represents the current context in the program and not the actual program counter register. A statement is only valid if the PC is *no more restrictive* than the involved variables of the statement. The PC label is calculated for each program block and is re-calculated at its entrance depending on the condition the block has been entered.

IV. Decentralized Label Model (DLM) for C Language (Cif)

During our application of the DLM to the C language we run into several challenges. The following sections provides design choices and implementation details on our implementation.

A. Type Checking Tool

The first step of our work was to define C annotations in order to apply DLM to this language. An annotated C program shall act as input for the DLM checker, in the following called *C Information Flow (Cif)*. Cif analyzes the program according to the defined information flow policy. Depending on the syntax of the annotations, the resulting C code can no longer be used as input for usual C compilers, such as the *gcc*. To still be able to compile the program, three major possibilities for implementing the Cif are available:

1) a Cif checking tool that translates the annotated input source code into valid C code by removing all labels
2) a DLM extension to available compilers, such as *gcc*
3) embedding labels into compiler-transparent comments using */* label */*

We decided for Option 1. We did not consider Option 2 to avoid necessary coding efforts for modifying and maintaining a specialized C compiler. We also did not take Option 3, due to the higher error-proneness resulting from the fact that our checker, additionally, had to decide whether a comment's content is a label or a comment. If a developer does not comply with the recognition syntax for labels, the checker could interpret actual labels as comments and omit their analysis. In the worst case the checker could vacuously report correct information flow for a program without carrying out any label comparison.

For being able to analyze the C source code statically, the first step in the tool chain is to resolve all macro definitions and to include the header files into one file. Fortunately, this step can be performed by using the *gcc*, since the compiler does not perform a syntax verification during the macro replacement. The resulting file then is used as input for our Cif checking tool. If Cif does not report any information flow violation, the tool will create a C-compliant source code by removing all annotations. Additional source code verifications, e.g., by Astrée [8], or the compilation for the final binary process on this plain C source file.

B. Syntax Extension of C Language

For the format and semantics of annotations, we decided to adapt the concepts of Jif [18], the DLM implementation for Java. So, we use curly brackets as token for the labels. For variable declarations, these labels have to be placed in between the type indicator and the name of the variable (cf. Listing 1). Compared to the reference implementation of Jif, in Cif we additionally had to deal with pointers of the C language. We annotate and handle pointers the same way as usual variables, i.e., when using a pointer to reference to an array element or other values, the labels of pointer and target variable have to match accordingly to DLM. However, Cif does not monitor overflows or invalid references whose detection calls for pointer calculations. We expect that additional tools (e.g., Astrée [8]) detect such coding errors. This tool is already used successfully for checking code of avionic equipment.

In addition to the new label tokens, we extended the syntax of the C language with five further tokens:

principal *p1, ..., pn*: This token announces all used principals to the Cif.

actsFor(*p, q*): This token statically creates a trust relation that principal *p* is allowed to act for principal *q* in the entire source code.

declassify(*variable*, {*label*}): This token allows to loosen a confidentiality policy in order to relabel variables if required. Cif checks whether the new confidentiality policy is less restrictive than the present one.

endorse(*variable*, {*label*}): This token allows to loosen an integrity policy in order to relabel variables if required. Cif checks whether the new integrity policy is less restrictive than the present one.

PC_bypass({*label*}): This token allows to relabel the PC label without further checks of correct usage.

C. Function Declaration

In the C language functions can have a separate declaration called prototype. For the declaration of functions and prototypes, we also adapted the already developed concepts from Jif. In Jif a method (the representation for a function in object-oriented languages) has four labels:

1) **Begin Label** defines the side effects of the function like accesses to global variables. The begin label is the initial PC label for the function's body. From a function caller's perspective the current caller's PC label needs to be *no more restrictive* than the begin label of the called function.
2) **Parameter Labels** define for each parameter the corresponding label. From a caller's perspective these parameter labels have to match with the assigned values.
3) **Return Label** defines the label of the return value of the function. In Cif, a function returning *void* cannot have a return label. From a caller's perspective the variable that receives the returned value needs to be at least equally restrictive as the return label.
4) **End Label** defines a label for the caller's observation how the function terminates. Since C does not throw exceptions and functions return equally every time, we omitted verifications of end labels in our Cif implementation.

Listing 4 shows the syntax for defining a function prototype with label annotations in Cif.

The definition of function labels regarding their optional prototype labels needs to be at least as restrictive, i.e., Cif allows functions to be more restrictive than their prototypes. All labels are optional augmentation to the C syntax. If the developer does not insert a label, Cif will use meaningful default labels that basically define the missing label most restrictively. Additionally, we implemented a label inheritance, which allows to inherit the real label of a caller's parameter value to the begin label, return label or other parameter labels of the function. This feature is useful for the annotation of

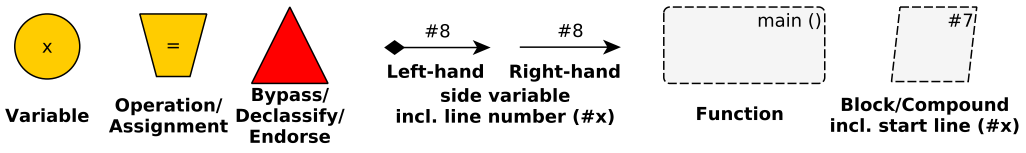

x
Variable
=
Operation/ Assignment
Bypass/ Declassify/ Endorse
#8
#8
Left-hand
Right-hand
side variable incl. line number (#x)
main ()
Function
#7
Block/Compound incl. start line (#x)

Fig. 1. Legend for Flow Graphs

$$\texttt{int}\ \overbrace{\{\texttt{Alice->Bob}\}}^{return\ label}\ \texttt{func}\ \overbrace{\{\texttt{param}\}}^{inherited\ begin\ label}\ (\texttt{int}\ \overbrace{\{\texttt{Alice->*}\}}^{parameter\ label}\ \texttt{param})\ :\ \overbrace{\{\texttt{Alice->*}\}}^{end\ label};$$

Listing 4. Definition of a function with DLM annotations in Cif.

system library functions, such as *memcpy(...)* that are used by callers with divergent parameter labels and can have side effects on global variables. At this stage Cif does not support the full inheritance of parameter labels to variable declarations inside the function's body.

D. *Using System Libraries*

Developers use systems libraries in their applications not only for convenience (e.g., to avoid reimplementation of common functionality) but also to perform necessary interaction with the runtime environment and the underlying operating system.

Hence, the system library provides an interface to the environment of the application, which mostly is not under the assurance control of the application's programmer. However, the code executed by library functions can heavily affect and also violate an application's information flow policy. Consequently, a system library needs to provide means for its functions to express the applied information flow policy and evidences to fully acknowledge this policy internally. In the best case, these evidences are also available by using our DLM approach. For Jif, the developers have annotated parts of the Java system library with DLM annotations that provide the major data structures and core I/O operations. Unfortunately, these annotations and its checks applied to all library functions demand many working hours and would exceed the available resource of many C development projects and, in particular, this research study. Luckily, other methods are conceivable, e.g., to gain evidences by security certification efforts of the environment. For our use case, the system software (a special Separation Kernel) was under security certification at the time of this study. Assuming the certification will be successful, we can assume its internals behave as specified. Furthermore, the research community worked on the formal specification and verification of Separation Kernels intensively, allowing us to trust the kernel if such methods have been applied [19], [20], [21]. However, we still had to create a special version of the system library's header file. This header file contains DLM-annotated prototype definitions of all functions of the Separation Kernel's system library. The Cif checker takes this file as optional input.

V. Use Cases

This section demonstrates the power of Cif by explaining usually appearing code snippets. For all examples Cif verifies the information flow modeled with the code annotations. If the information flow is valid according to the defined policy, Cif will output an unlabeled version of the C source code and a graphical representation of the flows in the source code. The format of this graphical representation is "graphml", hence, capable of further parsing and easy to import into other tools as well as documentation. Figure 1 shows the used symbols and their interpretations in these graphs. In general, the # symbol and its following number indicates the line of the command's or flow's implementation in the source code.

A. *Direct Assignment*

Listing 5 presents the first use case with a sequence of normal assignments.

```
principal Alice, Bob, Chuck;

void main {_->_;*<-*} ()
{
  int {Alice->Bob, Chuck} x = 0;
  int {Alice->Bob} y;
  int {Alice->*} z;

  y = x;
  z = y;
  z = x;
}
```

Listing 5. Sequence of Valid Direct Flows

In this example x is the least restrictive variable, y the second most restrictive variable and z the most restrictive variable. Thus, flows from $x \rightarrow y$, $y \rightarrow z$ and $x \rightarrow z$ are valid. Cif verifies this source code successfully and create the graphical flow representation depicted in Figure 2.

B. *Indirect Assignment*

Listing 6 shows an example of invalid indirect information flow. Cif reports an information flow violation, since all flows in the compound environment of the true if statement need to be at least as restrictive as the label of the decision variable

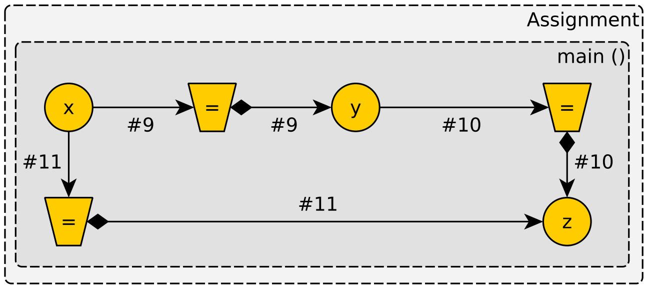


Fig. 2. Flow Graph for Listing 5

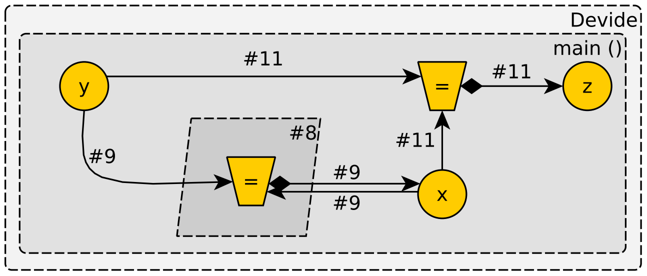


Fig. 3. Flow Graph for Listing 7

z. However, x and y are less restrictive and, hence, a flow to x is not allow. Additionally, this example shows how Cif can detect coding mistakes. It is obvious that the programmer wants to prove that y is not equal to 0 to avoid the Divide-by-Zero fault. However, the programmer puts the wrong variable in the if statement. Listing 7 corrects this coding mistake. For this source code, Cif verifies that the information flow is correct. Additionally, it generates the graphical output shown in Figure 3.

```
1   principal Alice, Bob;
2
3   void main {_->_;*<-*} ()
4   {
5     int {Alice->Bob} x, y;
6     int {Alice->*} z = 0;
7
8     if(z != 0) {
9       x = x / y;
10    }
11    z = x;
12  }
```

Listing 6. Invalid Indirect Flow

```
1   principal Alice, Bob;
2
3   void main {_->_;*<-*} ()
4   {
5     int {Alice->Bob} x, y;
6     int {Alice->*} z = 0;
7
8     if(y != 0) {
9       x = x / y;
10    }
11    z = x;
12  }
```

Listing 7. Valid Indirect Flow

Remarkable in Figure 3 is the assignment operation in line 9, represented inside the block environment of the if statement but depending on variables located outside of the block. Hence, Cif parses the code correctly. Also note, in the graphical representation z depends on input of x and y, even if the source code only assigns x to z in line 11. This relation is also depicted correctly, due to the operation in line 9 on which y influences x and, thus, also z indirectly.

Another valid indirect flow is shown in Listing 8. Interesting on this example is the proper representation of the graphical output in Figure 4. This output visualizes the influence of z on the operation in the positive if environment, even if z is not directly involved in the operation.

```
1   principal Alice, Bob;
2
3   void main {_->_;*<-*} ()
4   {
5     int {Alice->Bob} x, y, z;
6
7     if(z != 0) {
8       x = x + y;
9     }
10  }
```

Listing 8. Valid Indirect Flow

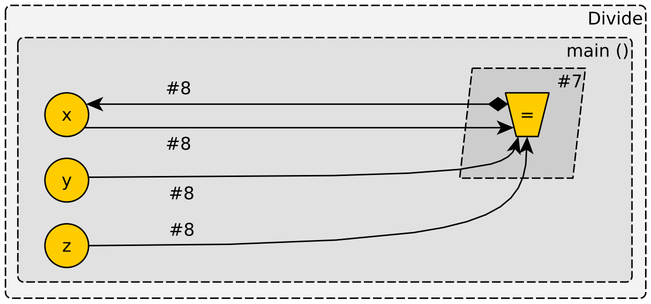


Fig. 4. Flow Graph for Listing 8

C. Function Calls

A more sophisticated example is the execution of functions. Listing 9 shows a common function call using the inheritance of DLM annotations. Line 3 declares the function. The label `{a}` signals the DLM interpreter to inherit the label of the declared parameter when calling the function; i.e., the label of parameter a for both, the label of parameter b and the return label. Essentially, this annotation of the function means that the data labels keep their restrictveness during the execution of the function. Line 14 and line 15 call the function twice with different parameters. The graphical representation of this flow in Figure 5 identifies the two independent function calls by the different lines of the code in which the function and operation is placed.

```
1   principal Alice , Bob;
2
3   float {a} func (int {Alice->Bob} a,
        float {a} b)
4   {
5     return a + b;
6   }
7
8   int {*->*} main {_->_} ()
9   {
10    int {Alice->Bob} y;
11    float {Alice->Bob} x;
12    float {Alice->*} z;
13
14    x = func(y,x);
15    z = func(y,0);
16
17    return 0;
18  }
```

Listing 9. Valid Function Calls

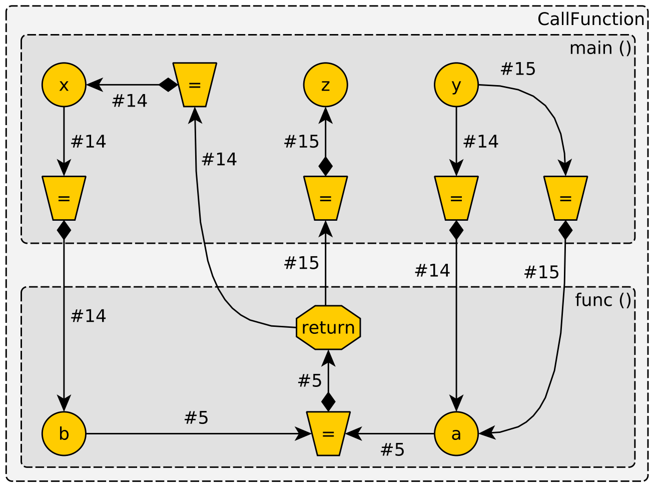


Fig. 5. Flow Graph for Listing 9

D. Declassify, Endorse and Bypassing the PC

1) Using Declassify and Endorse: Strictly adhering to the basic rules of DLM incurs the label-creeping problem [13]; the developer has to make information flow to more and more restrictive destinations. This unavoidably leads to the situation that information will be stored in the most restrictive variable and is not allowed to flow to some lower restricted destinations. Hence, sometimes developers need to manually declassify (for confidentiality) or endorse (for integrity) variables in order to make them usable for some other parts of the program. These intended breaches in the information flow policy need special care in code reviews and, hence, it is desirable that our Cif allows the identification of such sections in an analyzable way. Listing 10 provides an example using both, the endorse and declassify statement. To allow an assignment of a to b in line 9 an endorsement of the information stored in a is necessary. The destination b of this flow is less restrictive in its integrity policy than a, since `Alice` restricts `Bob` to not modify b anymore. In line 10, we perform a similar operation with the confidentiality policy. The destination c is less restrictive than b, since `Alice` believes for b that `Bob` cannot read the information, while `Bob` can read c.

The graphical output in Figure 6 depicts both statements correctly, and marks them with a special shape and color in order to attract attention to these downgrading-related elements.

```
1   principal Alice , Bob;
2
3   void main {_->_;*<-*} ()
4   {
5     int {Alice ->*; Alice<-Bob} a;
6     int {Alice ->*; Alice <-*}   b;
7     int {Alice->Bob; Alice <-*} c;
8
9     b = endorse(a, {Alice ->*;
                        Alice <-*});
10    c = declassify(b, {Alice->Bob;
                        Alice <-*});
11  }
```

Listing 10. Endorse and Declassify

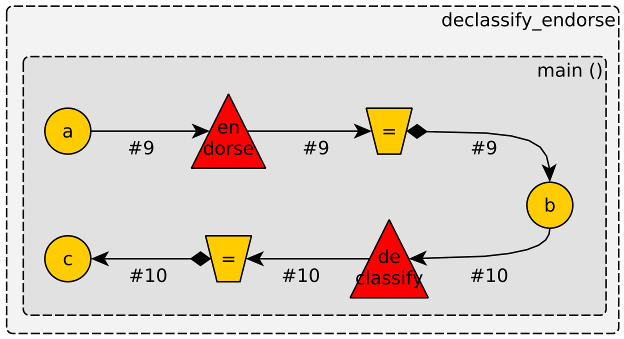


Fig. 6. Flow Graph for Listing 10

2) Bypassing the PC label: In the example of Listing 11 we use a simple login function to prove a user-provided uID and $pass$ against the stored login credentials. If the userID and the password match, a global variable $loggedIn$ is set to 1 to signal other parts of the application that the user is logged-in. The principal $System$ owns this status variable and represents the only reader of the variable. The principal $User$ owns both input variables uID and $pass$. The interesting lines of this example are lines 16–18, i.e., the conditional block that checks whether the provided credentials are correct and change the status variable $loggedIn$. Note, that this examples also presents Cif's treatment of pointers on the `strcmp` function. Due to the variables in the boolean condition of the *if* statement, the PC label inside the following block is `System-> & User->`. However, this PC is not more restrictive than the label of *loggedIn* labeled with `System->`. Hence, Cif would report an invalid indirect information flow on this line. To finally allow this light and useful violation of the information flow requirement, the programmer needs to manually downgrade or bypass the PC label as shown in line 17. In order to identify such manual modifications of the information flow policy, Cif also adds this information in the generated graphical representation by using a red triangle indicating the warning (see Figure 7). This shall enable code reviewers to identify the critical sections of the code to perform their (manual) review on these sections intensively.

```
1   principal User, System;
2
3   int {System->*} loggedIn = 0;
4
5   int {*->*} strcmp {*->*}
        (const char {*->*} *str1,
         const char {*->*} *str2)
6   {
7     for(; *str1==*str2 && *str1; str1++,
          str2++) ;
8     return *str1 - *str2;
9   }
10
11  void checkUser {System->*}
        (const int {User->*} uID,
         const char {User->*} * const pass)
12  {
13    const int {System->*} regUID = 1;
14    const char {System->*} const
          regPass[] = "";
15
16    if(regUID == uID &&
          !strcmp(regPass, pass)) {
17      PC_bypass({System->*});
18      loggedIn = 1;
19    }
20  }
```

Listing 11. Login Function

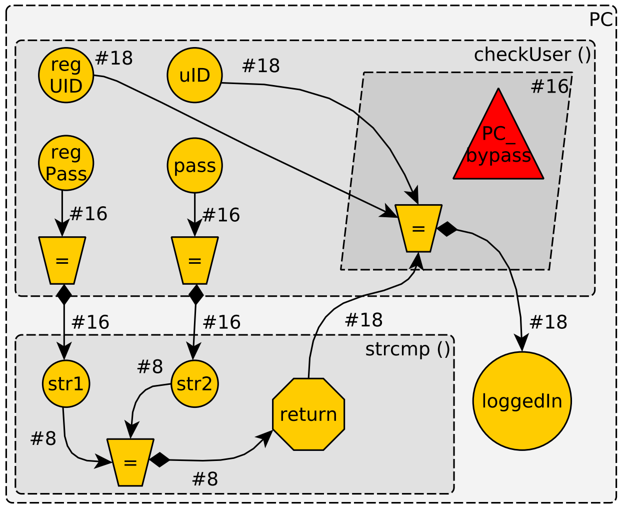


Fig. 7. Flow Graph for Listing 11

VI. Use-Case: The Avionics Security Domains

Due to their diversity in functions and criticality on the aircraft's safety, on-board networks are divided into security domains. The ARINC standards (ARINC 664 [22] and ARINC 811[23]) define four domains also depicted in Figure 8:

1. **Aircraft Control:** The most critical domain hosting systems that support the safe operation of the aircraft, such as cockpit displays and system for environmental or propulsion control. This domain provides information to other (lower) domains but does not depend on them.
2. **Airline Information Services:** This domain acts as security perimeter between the Aircraft Control Domain and lower domains. Among others it hosts systems for crew information or maintenance.
3. **Passenger Information and Entertainment Services:** While being the most dynamic on-board domain regarding software updates, this domain hosts systems related to the passenger's entertainment and other services such as Internet access.
4. **Passenger-owned Devices:** This domain hosts mobile systems brought on-board by the passengers. They may connect to aircraft services via an interface of the Passenger Information and Entertainment Services Domain.

To allow information exchange between those domains, additional security perimeters have to be in place to control the data exchange. Usually, information can freely flow from higher critical domains to lower critical domains. However, information sent by lower domains and processed by higher domains need to be controlled. This channel is more and more demanded, e.g., by the use case of the maintenance interface that is usually hosted within the Airline Information Service Domain but also should be used for updating the Aircraft Control Domain. For protecting higher domains from the threat of vulnerable data a security gateway can be put in service in order to assure integrity of the higher criticality domains. This security gateway examines any data exchange and assures integrity of the communication data and consequently of the high integrity domain. Since this gateway is also a highly critical system, it requires similar design and implementation assurances regarding safety and security as the systems it protects.

VII. Use-Case: The Railway Security Domains

The railway industry needs to protect the integrity and availability of their control network, managing signals, positions of trains and driving parameters of trains. Hence, also the

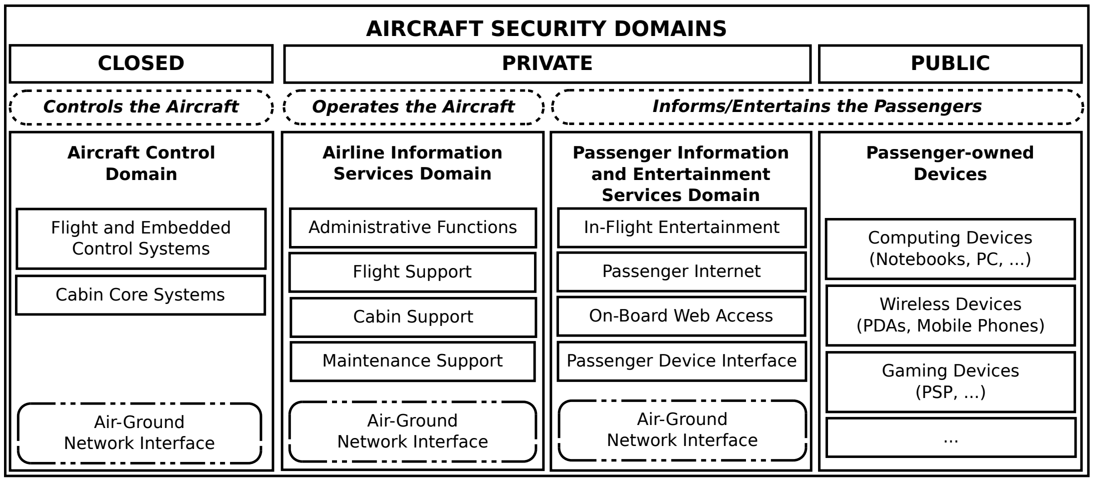

AIRCRAFT SECURITY DOMAINS
CLOSED
PRIVATE
PUBLIC
Controls the Aircraft
Operates the Aircraft
Informs/Entertains the Passengers
Aircraft Control Domain
Flight and Embedded Control Systems
Cabin Core Systems
Air-Ground Network Interface
Airline Information Services Domain
Administrative Functions
Flight Support
Cabin Support
Maintenance Support
Air-Ground Network Interface
Passenger Information and Entertainment Services Domain
In-Flight Entertainment
Passenger Internet
On-Board Web Access
Passenger Device Interface
Air-Ground Network Interface
Passenger-owned Devices
Computing Devices (Notebooks, PC, ...)
Wireless Devices (PDAs, Mobile Phones)
Gaming Devices (PSP, ...)
...

Fig. 8. Avonic Security Domains as defined by ARINC 664 [22] and ARINC 811 [23].

railway industry has categorized their systems and interfaces into security domains.

Railway control consists of several domains from control centers over interlocking systems to field elements all interacting in one way or the other with onboard systems. For interlocking, DIN VDE V 0831-104 [24] defines a typical architecture from a security zone perspective, which is depicted in Figure 9.

For interlocking, Figure 9 shows that different levels of maintenance and diagnosis are needed. Local maintenance interacts via a gateway (demilitarized zone) with control elements interlocking logic, operator computers and field elements Considering in this example the diagnosis information, the diagnosis database needs a method for data acquisition without adding risks of propagating data into the interlocking zone. To implement this, a simple diode-based approach is deemed sufficient. Remote diagnosis is more complicated with access to diagnosis as well as the interlocking zone, but again using gateways to access control elements (interlocking, operator, and field element computers).

This example of accessing interlocking for diagnosis and maintenance purposes reflects the potential need for security gateways. In case of operation centers where many interlockings are controlled and monitored remotely, similar security measure are to be taken if connected via open networks. Similarly if within different interlockings communication runs over open networks, encryption and potentially also gateway approaches may be needed.

In current and future signalling, control and train protection systems such as European Train Control System (ETCS) level 2 or higher security aspects need to consider aspects of wireless communication and – similarly to approaches described above – need to protect different system components and systems.

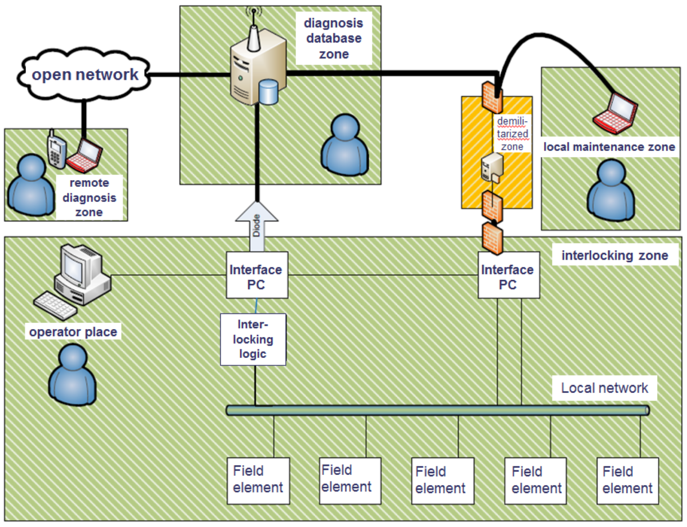


Fig. 9. Railway Security Zones [24]

VIII. The MILS Gateway

The avionic and railway use case share one major commonality regarding security. Both industries elaborated security classifications for their systems; depending on the criticality and users systems are categorized into security domains. However, systems of these security domains mostly cannot operate independent but often demand data from systems of other domains. For example, services of the avionic Passenger Information and Entertainment Services domain need data from systems of the Aircraft Control domain, such as the altitude and position of the aircraft for enabling or disabling the on-board WiFi network due to regulations by governmental authorities. In railway an example for data exchange is the external adjustment of the maximum allowed train speed, triggered by the train network operator. To still protect a domain against invalid accesses or malicious data, control instances such as Secure Network Gateways are deployed. These gateways mediate and control the data exchange on the domain borders and filter the data according to a defined information flow policy.

In previous work we have introduced a Secure Network Gateway [25] based on the MILS approach.

The MILS architecture, originally introduced in [26] and further refined in [5], is based on the concept of spatial and temporal separation connected with controlled information flow. These properties are provided by a special operating system called a Separation Kernel. MILS systems enable one to run applications of different criticality integrated together on one hardware platform.

Leveraging these properties the gateway is decomposed into several subfunctions that operate together in order to achieve the gateway functionality. Figure 10a shows the partitioned system architecture. The benefit of the decomposition is the ability to define local security policies for the different gateway components. The system components themselves run isolated among each other within the provided environments of a Separation Kernel. Using a Separation Kernel as a foundational operating system guarantees non-interference between the identified gateway subfunctions except when an interaction is granted. Hence, the Separation Kernel provides a coarse information flow control in order to prove which component is allowed to communicate to which other component. However, within the partition's boundaries the Separation Kernel cannot control the correct implementation of the defined local information flow policy. This paper presents a new concept of connecting MILS with the DLM in order to fill this gap and to provide system-wide evidence of correct information flows.

In comparison to our unidirectional gateway of [25] comprising just two partitions to perform information flow control on very basic protocols only, our improved gateway is composed of four major logical components (cf. also Figure 10a):

1) The Receiver Components
2) Filter Component(s)*
3) The Transmitter Components
4) Health Monitoring and Audit Component*

* (not depicted in Figure 10a)

Figure 10b extracts the internal architecture of the Receiver Component being a part of our gateway system. The task of this component is to receive network packets from a physical network adapter, to decide whether the packet contains TCP or UDP data, and to parse and process the protocols accordingly. Hence, this component is composed of three subfunctions hosted in three partitions[2] of the Separation Kernel:

1) **DeMux:** Receiving network packets from the physical network adapter, and analyzing and processing the data traffic on lower network protocol levels (i.e., Ethernet/-MAC and IPv4[3])
2) **TCP_Decoder:** Analysing and processing of identified TCP packets
3) **UDP_Decoder:** Analysing and processing of identified UDP packets

[2]A *partition* is a runtime container in a Separation Kernel that guarantees non-interfered execution. A *communication channel* is an a priori defined means of interaction between a source partition and one or more destination partitions.

[3]For the following we assume our network implements Ethernet and IPv4, only.

The advantage of this encapsulation of subfunctionality into three partitions is the limitation of possible attack impacts and fault propagation. Generally, implementations of TCP stacks are considered more vulnerable to attacks than UDP stacks, due to the increased functionality of the TCP protocol compared to the UDP protocol. Hence, the TCP stack implemented in the TCP_Decoder can be assumed as more vulnerable. A possible attack vector to a gateway application is to attack the TCP stack in order to circumvent or to perform denial-of-service on the gateway. If all three subfunctions run inside one partition, the entire Receiver Component would be affected by a successful attack on the TCP stack. However, in this distributed implementation using the separation property of the Separation Kernel only the TCP_Decoder would be affected by a successful attack. A propagation of the attack impact (or fault) to the UDP_Decoder or DeMux is limited due to the security properties of the Separation Kernel.

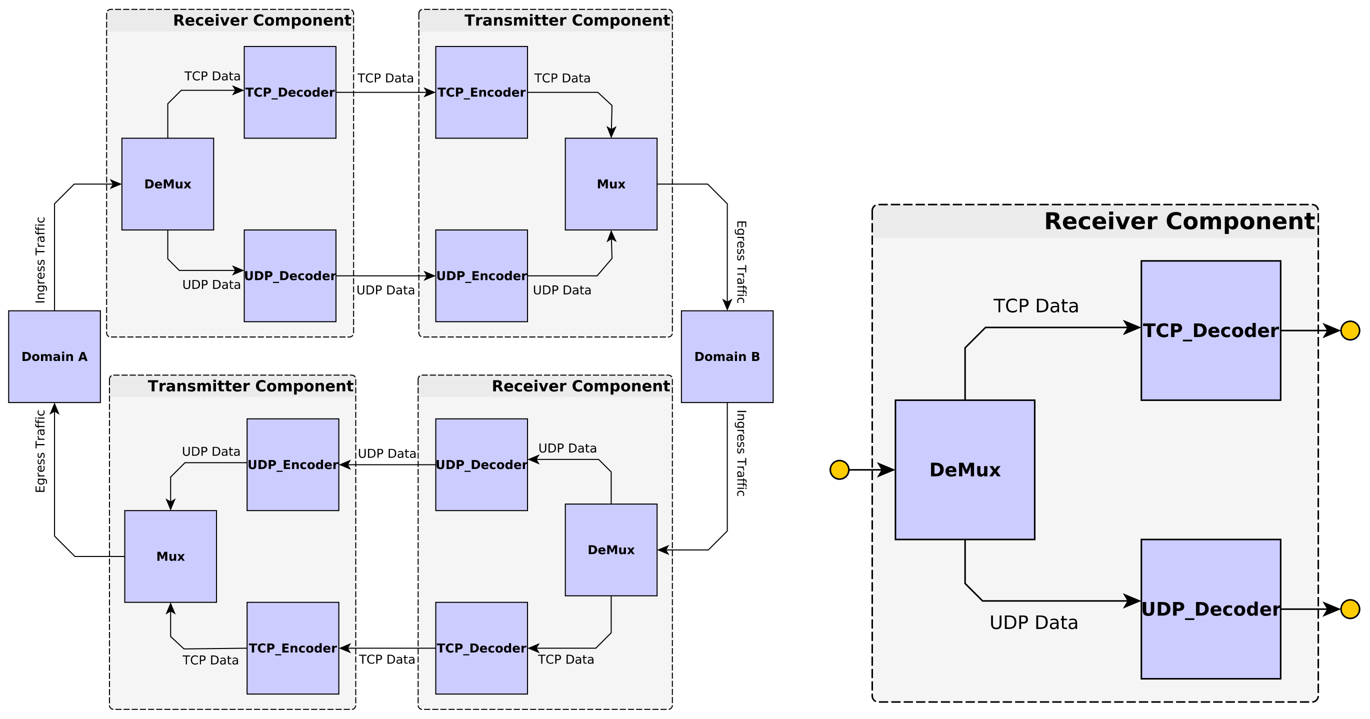
Further developing the gateway example, the strength of using DLM is to assure a correct implementation of the demultiplexer running in the DeMux partition. Considering the C code in Listing 12 the essential part of the demultiplexer requires the following actions:

- Line 2 and line 3 define the prototypes of the functions that send the data to the subsequent partitions using either channel *TCP data* or channel *UDP data* of Figure 10b.
- Line 5 defines the structure of the configuration array containing an integer value and a function-pointer to one of the previously defined functions.
- The code snippet following line 11 configures the demultiplexer by adding tuples for the TCP and UDP handlers to the array. The integer complies with the RFC of the IPv4 identifying the protocol on the transport layer.
- Line 29 implements the selection of the correct handler by iterating to the correct element of the configuration array and comparing the type field of the input packet with the protocol value of the configuration tuples. Note that the loop does not contain any further instructions due to the final ';'.
- The appropriate function is finally called by line 32.

A. DLM Applied to the Gateway Use Case

We consider again the use case presented in Figure 10b. In order to use DLM for the DeMux of the Receiver Component an annotation of Listing 12 is needed. Listing 13 shows this annotated version. The graphical representation is depicted in Figure 11.

- Line 1 announces all used principals of this code segment to the Cif checker.
- In line 3 and line 4 we label the begin label and the data parameter of the two prototype declarations with labels



(a) Global system architecture

(b) The Receiver Component is a function composed of three partitions.

Fig. 10. MILS Gateway Architecture. Blue boxes indicate partitions.
The Separation Kernel assures partition boundaries and communication channels (arrowed lines).

that either principal TCP or principal UDP owns the data starting with the time the function is called.

- The definition of the input buffer in line 13 receives also a label. For the confidentiality policy the data is owned by the *Ethernet*, since data will be received from the network and we assume it is an Ethernet packet. This owner *Ethernet* allows both *TCP* and *UDP* to read its data. The integrity policy of line 13 is a bit different. The top principal can act for all principals and, hence, the data is owned by all principals. However, all principals assume that only Ethernet has modified the data. This assumption is correct, since data is received from the network.
- The main function of our program (line 15) now contains a begin label. This label grants the function to have a side effect on the global INPUT variable (i.e., the input buffer).
- Due to our language extension we had to replace the more elegant *for* loop (cf. line 29 in Listing 12) by a *switch-case* block (cf. line 17)). Within each case branch, we have to relabel the data stored in INPUT in order to match the prototype labels of line 3 or line 4 accordingly. The first step of this relabeling is a normal information flow by adding *TCP* (or *UDP*) as owner to the confidentiality policy and integrity policy (cf. line 20 and line 28). This step is performed compliantly to the DLM defined in Section III-B. Then, we *bypass* the PC label in order to change to the new environment and to match the begin label of the associated decoder function (cf. line 21 and line 29). As a second relabeling step, we still need to remove the principal *Ethernet* from the confidentiality and integrity policies. Removing this principal does not comply with the allowed information flow of DLM, since both resulting policies are less restrictive than the source policies. Hence, we have to declassify (for the confidentiality policy) and endorse (for the integrity policy) by using the statements of line 22 and line 30.
- Finally we can call the sending functions in line 23 and line 31 accordingly.

```
1  principal Ethernet, UDP, TCP;
2
3  TCP_SendDecode {TCP->*; TCP<-*} (void {TCP
       ->*; TCP<-*} *data);
4  UDP_SendDecode {UDP->*; UDP<-*} (void {UDP
       ->*; UDP<-*} *data);
5
6  static union {
7    struct {
8      /* [...] further fields of protocols
           */
9      char protocol;
10     /* [...] further fields of protocols
           */
11   } u;
12   char buf[0xffff];
13 } {Ethernet->TCP, UDP; *<-Ethernet} INPUT;
14
15 int main {Ethernet->TCP, UDP; *<-Ethernet}
```

```
/* DECLARATION */
TCP_SendDecode(void *data);
UDP_SendDecode(void *data);

typedef struct {
  char protocol;
  void (*func)(void* data);
} DeMux;

/* CONFIGURATION */
static DeMux handler[] = {
  { 0x06, &TCP_SendDecode },  // 0x06 indicates TCP in IPv4
  { 0x11, &UDP_SendDecode },  // 0x11 indicates UDP in IPv4
  { 0x00, 0 }        // final entry
};

union { struct {
    /* [...] further fields of protocols */
    char protocol;
    /* [...] further fields of protocols */
  } u;
  char buf[0xffff];
} INPUT;

int main() {
  /* [...]  load data from network into INPUT */
/* PROCESSING */
  DeMux* itr = 0;
  for(itr = handler; itr->protocol != 0 &&
    itr->protocol != INPUT.u.protocol; itr++) ;
  if(itr->func != 0)
    (*itr->func)(INPUT);
  else
    Error();
}
```

Listing 12. Demultiplexer of the Receiver Component

```
     () {
16    /* [...]  load data from network into INPUT */
17    switch(INPUT.u.protocol) {
18      case 0x06:  /* 0x06 in IPv4 indicates TCP */
19      {
20        void {Ethernet & TCP->*; TCP<-Ethernet} *ptr = INPUT.buf;
21        PC_bypass({TCP->*; TCP<-*});
22        void {TCP->*; TCP<-*} *tcp = endorse(declassify(ptr, {TCP->*; TCP<-Ethernet}), {TCP->*; TCP<-*});
23        TCP_SendDecode(tcp);
24        break;
25      }
26      case 0x11:  /* 0x11 in IPv4 indicates UDP */
27      {
28        void {Ethernet & UDP->*; UDP<-Ethernet} *ptr = INPUT.buf;
29        PC_bypass({UDP->*; UDP<-*});
30        void {UDP->*; UDP<-*} *udp = endorse(declassify(ptr, {UDP->*; UDP<-Ethernet}), {UDP->*; UDP<-*});
31        UDP_SendDecode(udp);
32        break;
33      }
34      default:
35      {
36        Error();
37      }
38    }
39  }
```

Listing 13. Annotated Receiver Component

Clearly, Figure 11 indicated six warnings inside the source code by the red triangle. The bypass of the PC label in line line:cif-tcp-bypass and line line:cif-udp-bypass reason two of these warnings. The remaining four warnings are due to the endorse and declassification (two each) of the DLM information flow policy in order to allow the assignments in line 22 and line 30. As DLM provides information flow assurance, code reviewers just need to concentrate on these indicated sections of the code to perform manual validation. The remaining source code is validated thanks to the DLM

Using the presented technology we also annotate other critical gateway components hosted in other partitions of the MILS system. The *Cif* tool is able to process all needed C language features of our implementation, e.g., loops, decision branches, switch-case blocks, function calls, pointer arithmetics, and also function pointers as long as their DLM policy is homogeneous (cf. Section IX). Since the tool does not report information flow violations of the local defined policies, we gain additional evidence of the correct implementation of the gateway's components. Together with the defined MILS information flow policy ensured by the Separation Kernel we can assure a correct implementation of the system-wide information flow. Currently, we have still had to map both evidences manually for developing the prove of concept. Future planning of our implementation involves to automatically combine both steps.

IX. Enhancing the Decentralized Label Model

A. Assessment of DLM applied to the Gateway Use Case

The application of DLM to the gateway use case identified some advantages and disadvantages. The **advantages** are on easing the assurance process and on the improved identification of code flaws, being more detailed:

Automatic Proof of Correct Information Flows: The presented source code in Listing 13 complies with a formally proven information flow model, the DLM. The proof of this properties of complient information flows could be achieved automatically with tool support. Hence, it can be considered as highly assured that the present information flows are correct. The graphical representatio clearly identifies code sections that endorsing or declassifying the modeled information flow policy. This allows to reduce

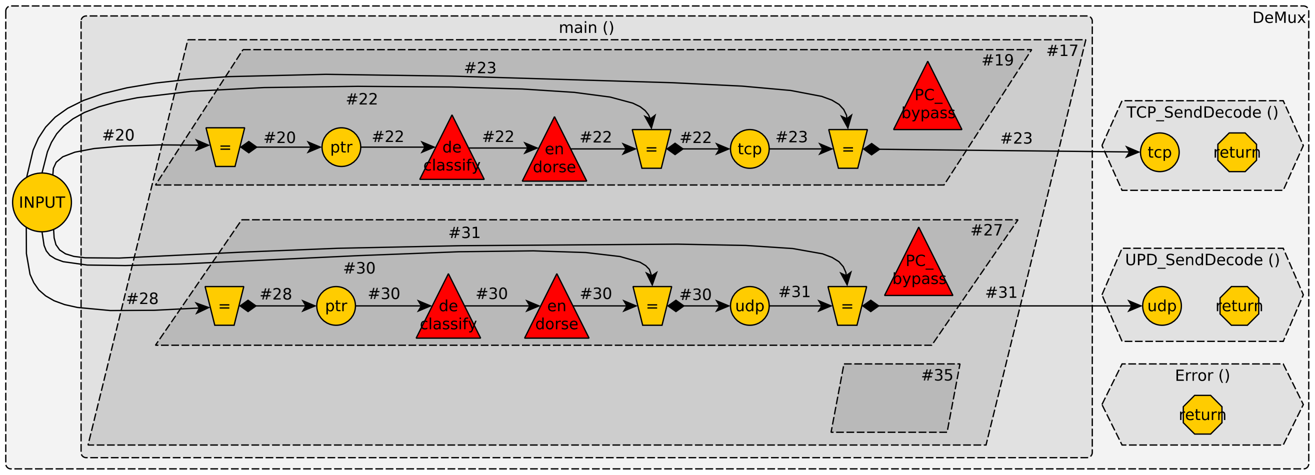


Fig. 11. DLM Information Flows in the Demultiplexer of the MILS Gateway implemented by Listing 13.

efforts for code reviews, as the reviewers can focus on these parts of the code. In contrast, all the code in Listing 12 requires detailed and manual code review in order to provide the same level of assurances.

Increased Robustness against Coding Flaws: Compared to Listing 12, the DLM-inducted code snippet of Listing 13 is more robust against coding flaws. Imagine a distracted programmer swaps the two function pointers in the configuration array of Listing 12. This modification will result in a probably hardly debugable runtime error, since UDP traffic will now be sent to the TCP_Decoder and the other way around. In contrast, if swapping the function calls of line 23 and line 31 of Listing 13 the Cif checker will raise an information flow breach due to the invalid labels of the function call and the labels of the parameter. Hence, this DLM allows detection of possible runtime errors before compilation.

Contrary, the following **disadvantages** have been identified:

Larger Code Size: Comparing both demultiplexer implementations of Listing 12 and Listing 13 it is obvious that the DLM-inducted version increases the code size due to the different employed programming styles. In the former, we use a table-based programming technique (see the *for* loop) whereas in the latter we use a code-based programming technique (see the *switch* statement). Another reasons for the bigger foot print is the relabeling directives within the *case* blocks. This relabeling introduces new variables and statements that are only required for inducting DLM. From a program control flow perspective, those statements are useless and (hopefully) detected and removed by the compiler during optimization.

Reduced Readability: The label annotations of DLM itself form another disadvantage. Their introduction may reduce readability of the code due to the unusuality of their use. However, using DLM increases the level of automation in providing assurance, and thus, the need of manual code review. The best case avoids manual code reading entirely.

Considering the identified disadvantages, we elaborate on a solution to reduce the code size by extending DLM to allow expressing DLM policies on the table-based programming technique of Listing 12.

B. Extensions to the DLM

Based on [27], [28] we will now describe a possible extension of Cif intended to overcome the shortcoming. Listing 14 displays the resulting code. It should be clear that it is rather close to that of Listing 12; a similar version could be developed for Listing 13.

```
/* DECLARATION */
principal Ethernet, UDP, TCP;

policy GatewayHandler
= (self.protocol==0x06 =>
 self.func=={TCP->*; TCP<-*} (void {TCP->*; TCP<-*} *))
&& (self.protocol==0x11 =>
 self.func=={UDP->*; UDP<-*} (void {UDP->*; UDP<-*} *));

policy Gateway
= (self.protocol==0x06 => self=={TCP->*; TCP<-*})
&& (self.protocol==0x11 => self=={UDP->*; UDP<-*});

TCP_SendDecode {TCP->*; TCP<-*} (void {TCP->*; TCP<-*} *data);
UDP_SendDecode {UDP->*; UDP<-*} (void {UDP->*; UDP<-*} *data);

typedef struct {
  char protocol;
  void (*func)(void* data);
} {GatewayHandler} DeMux;
```

```
21
22  /* CONFIGURATION */
23  static DeMux {GatewayHandler} handler[] =
        {
24    { 0x06, &TCP_SendDecode },
25    { 0x11, &UDP_SendDecode },
26    { 0x00, 0}
27  };
28
29  union {
30    struct {
31      /* [...] further protocol fields */
32      char protocol;
33      /* [...] further protocol fields */
34    } u;
35    char buf[0xffff];
36  } {Gateway} INPUT;
37
38  int main() {
39    /* [...]  load data from network into
          INPUT */
40    /* and if necessary declassify to give
          it the right type */
41
42    /* PROCESSING */
43    DeMux* itr = 0;
44    for(itr = handler; itr->protocol != 0x00
          && itr->protocol != INPUT.u.
         protocol; itr++) ;
45    if (itr->protocol != 0)
46      (*itr->func)(INPUT);
47    else
48      /* Neither a TCP nor UDP packet -->
          ERROR */
49  }
```

Listing 14. Annotated Receiver Component in Enhanced DLM

The basic idea is to extend DLM with policies that depend on the actual values of data. Consider the Gateway policy defined in line 10 onwards. The intention is that the entire field should obey the policy `{TCP->*; TCP<-*}` in case the protocol component (INPUT.u.protocol) equals 0x06; it should obey the policy `{UDP->*; UDP<-*}` in case the protocol component equals 0x11; and if the protocol component has a value different from 0x06 and 0x11 no requirements are imposed on the entire field.

The general form of the syntax used is to let policies be constructed according to the following grammar:

$$\begin{array}{rcl}
\textit{policy} & ::= & \textit{field}{=}{=}\textit{DLMpolicy} \\
 & | & \textit{policy}\ \texttt{\&\&}\ \textit{policy} \\
 & | & \textit{policy}\ \texttt{||}\ \textit{policy} \\
 & | & \textit{condition}\ \texttt{=>}\ \textit{policy} \\
 & | & \textit{condition}\ \texttt{\&\&}\ \textit{policy} \mid \cdots \\
\textit{condition} & ::= & \textit{field}{=}{=}\textit{value} \\
 & | & \textit{condition}\ \texttt{\&\&}\ \textit{condition} \mid \cdots \\
\textit{field} & := & \texttt{self} \mid \textit{field.component} \\
\textit{DLMpolicy} & := & \text{as previously used but extended to} \\
 & & \text{function types}
\end{array}$$

Here `=>` denotes implication, `&&` denotes conjunction, and `||` denotes disjunction. The identifier `self` is a reserved token for the data structure in question and *component* lists possible components.

To make use of such extended policies one needs to track not only the DLM policies and the types pertaining the data but also to track the information about the values of data that can be learnt from the various tests, branches and switches being performed in the program. The development in [27][28] achieves this by combining a Hoare logic for tracking the information about the values of data with the DLM policies and allows us to validate the code snippet in Listing 14.

This suffices for solving two shortcomings discussed above. First, it reduces the need to use declassification and PC_bypass for adhering to the policy thereby reducing the need for detailed code inspection. Second, it permits a more permissive programming style that facilitates the adoption of our method by programmers.

From an engineering point of view, the ease of use of conditional policies are likely to depend on the style in which the conditional policies are expressed. The development in [27] considers policies that in our notation would be written in the form of policies in Disjuntive Normal Form (using `||` at top-level and `&&` at lower levels), whereas the development in [28] considers policies that in our notation would be written in the form of policies in Implication Normal Form (using `&&` at top-level and `=>` at lower levels). The pilot implementation in [29], [30] seems to suggest that forcing policies to be in Implication Normal Form might be more intuitive and this is likely the way we will be extending Cif.

From an expressiveness point of view, it does not matter whether one uses Disjunctive Normal Form or Implication Normal Form. For example, we might consider to change the Gateway policy to the more demanding policy

```
policy Gateway
= (self.u.protocol==0x06 && self=={TCP->*;
    TCP<-*})
 || (self.u.protocol==0x11 && self=={UDP->*;
    UDP<-*})
```

expressing that there are no other permitted possibilities for the protocol component than to be either 0x06 or 0x11. This is desirable for the code snippet illustrated because line 48 would then not be reachable; however, it may be harder to ensure that the INPUT received from the network adheres to this policy. While the Disjunctive Normal Form expresses this, we could obtain the same result in Implication Normal Form by writing

```
policy Gateway
= (self.u.protocol==0x06 => self=={TCP->*;
    TCP<-*})
&& (self.u.protocol==0x11 => self=={UDP->*;
    UDP<-*})
&& (self.u.protocol!=0x06 && self.u.protocol
    !=0x11 => self=={Z->Z;Z<-Z})
```

where Z is an otherwise unused principal and hence the policy $= Z \to Z; Z \leftarrow Z$ is unattainable.

A final problem that would need to be overcome is how to interface the checking of conditions to the programmer. In the pilot implementation reported in [29][30] it is demanded that the programmer provides invariants for all while loops (since in general this is undecidable).

We are therefore experimenting with ways of using the results of the powerful static analyser Astree to provide the required invariants and possibly to use the abstract properties of Astree in expressing the policies. Current results [31] suggest that this approach to be very promising, which would make it a strong candidate for inclusion into Cif.

X. Conclusion

In this article, we presented C Information Flow (Cif), a static verification tool to check information flows modeled directly in C source code. Cif is an implementation of the Decentralized Label Model (DLM) [10] for the programming language C. To the best of our knowledge, we applied DLM to the C language for the first time. During the application of DLM to C, we tried to stick to the reference implementation of Java/Jif. However, we had to solve some language-specific issues, such as pointer arithmetic or the absence of exceptions. Additionally, we added the possibility of defining annotations to function prototypes only, in case a library's source code is not available for public access. Then, we also introduced rules for differing annotations of function prototypes compared to function implementations.

In various code snippets, we discussed information flows as they appear commonly in C implementations. Cif is able to verify all of these examples successfully. In case of valid information flows through the entire source code, Cif generates a graphical representation of the occurring flows and dependencies — a distinguishing feature not possessed by Jif. This graphical representation covers direct assignments of variables, logical and arithmetic operations, indirect dependencies due to decision branches and function calls. Cif allows the programmer to make declassifications and endorsements as in DLM, and additionally marks the places where flow policies are loosened with declassifications and endorsements in the graphical representation. Since DLM-annotated source code shall reduce the efforts of manual code reviews, these graphical indications allow to identify critical parts of the source code. Such parts usually require then special investigation during code reviews.

Further on, we presented how the security-typed language system [13] of the DLM can be connected to Multiple Independent Levels of Security (MILS) systems. MILS [26] is a system architecture to build high-assurance systems [5]. Various industrial domains require such high-assurance systems to fulfill special safety and security demands. MILS bases on the properties of separation and controlled information flow, both provided by a special kind of operating system, called a Separation Kernel. Such kernels provide separated runtime environments to host applications and to assure a configurable information flow policy among those environments. However, the Separation Kernel is not able to control the internals of these runtime environments. Security-typed languages such as the DLM can fill this gap.

In our use case, we target the example of a gateway application. In our study, we have identified this use case as a common implementation challenge for high assurance systems from the avionics and railway industry; however, our approach is not limited to those two industries but is also conceivable for other industries such as smart meters or automotive. This gateway supervises the information exchange between security domains, either on-board aircraft or between a train and its railway operational control network. Architecturally, the gateway follows the design principles of MILS. To control the system's information flows we connected the coarse information flows assured by the Separation Kernel with the application-dependent information flows, expressed by DLM-annotated C source code of the gateway's implementation. Compared to other security-typed languages for C, e.g., as proposed by Greve [14] using a mandatory access control-based approach, we used a decentralized approach for assuring correct information flow. This has the advantage of revealing subtler unwwanted dependencies in code, and explicating the mutual distrust between different software components. The latter also provides more flexibility in modeling the information flow policy.

We applied DLM annotations to a typical security function for high assurance systems: a demultiplexer which is part of the MILS-based gateway application. Using our developed Cif, we are able to ensure secure information flows within the gateway's components according to the defined information flow policy. Particularly, the visualization of indirect flows, e.g., Listing 7 or Listing 8, and function calls, e.g., Listing 9, were very useful during the evaluation of the use case. Additionally, this activity showed that Cif is able to cover larger projects, too. Connecting DLM proofs with the information flow assurance of the Separation Kernel provides system-wide evidences of correct implementation, e.g., as required by high Evaluation Assurance Levels of Common Criteria certifications. However, to annotate source code using the current model of DLM implemented by our Cif required to change parts of the source code. Using the presented technology we annotated further critical parts of our gateway to prove their correct implementation.

Additionally, we evaluated the benefits and drawbacks of applying DLM to C. While the benefits are clearly in the automation of gaining assurances of correct code and the reduction of manual code review, the drawbacks are the usability and increased code's footprint. Both disadvantages are critical for the future developer's acceptance of our approach and will finally decide on whether this DLM assurance can be successful in a wider field of application. To improve usability, we proposed an enhancement to the DLM, theoretically rooted in

[27], that removes the need of the presented code modification. We implemented a proof-of-concept checker [29] in parallel to Cif, to assure correct information flows within this new DLM-aware theory. Compared with Cif, this prototype checker does not support rich language features or the generation of graphical representations at the current stage.

As future work, we will evaluate on merging the features of Cif with the extended assurance of this prototyped implementation.

Acknowledgment

This work was supported by the ARTEMiS Project SeSaMo, the European Union's 7^{th} Framework Programme project EURO-MILS (ID: ICT-318353), the German BMBF project SiBASE (ID: 01IS13020) and the project EMC2 (grant agreement No 621429, Austrian Research Promotion Agency (FFG) project No 84256,8 and German BMBF project ID 01IS14002). We also thank Kawthar Balti, Tomasz Maciazek and Andre Ryll for their input.

References

[1] K. Müller, S. Uhrig, M. Paulitsch, and S. Uhrig, "Cif: A Static Decentralized Label Model (DLM) Analyzer to Assure Correct Information Flow in C," in *Proc. of the 10^{th} International Conference on Software Engineering Advances (ICSEA 2015)*. Barcelona, Spain: IARIA, Nov. 2015, pp. 369–375. [Online]. Available: http://www.thinkmind.org/index.php?view=article&articleid=icsea_2015_14_20_10272

[2] EUROCAE/RTCA, "ED-12C/DO-178C: Software Considerations in Airborne Systems and Equipment Certification," European Organisation for Civil Aviation Equipment / Radio Technical Commission for Aeronautics, Tech. Rep., 2012.

[3] ——, "ED-80/DO-254, Design Assurance Guidance for Airborne Electronic Hardware," European Organisation for Civil Aviation Equipment / Radio Technical Commission for Aeronautics, Tech. Rep., 2000.

[4] H. Butz, "The Airbus Approach to Open Integrated Modular Avionics (IMA): Technology, Methods, Processes and Future Road Map," Hamburg, Germany, March 2007.

[5] J. Rushby, "Separation and Integration in MILS (The MILS Constitution)," SRI International, Tech. Rep. SRI-CSL-08-XX, Feb. 2008.

[6] J. Alves-Foss, W. S. Harrison, P. W. Oman, and C. Taylor, "The MILS Architecture for High-Assurance Embedded Systems," Tech. Rep., 2006.

[7] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A Practical Tutorial on Modified Condition/Decision Coverage," NASA, Tech. Rep. May, 2001. [Online]. Available: http://shemesh.larc.nasa.gov/fm/papers/Hayhurst-2001-tm210876-MCDC.pdfhttp://dl.acm.org/citation.cfm?id=886632

[8] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, A. Miné, X. Rival, L. Mauborgne, A. Angewandte, I. Gmbh, S. Park, and D. Saarbrücken, "Astrée: Proving the Absence of Runtime Errors," in *Proc. of the Embedded Real Time Software and Systems (ERTS2'10)*, Toulouse, France, 2010, pp. 1–9.

[9] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: http://dl.acm.org/citation.cfm?id=360248.360252

[10] A. C. Myers and B. Liskov, "A Decentralized Model for Information Flow Control," in *Proc. of the 16^{th} ACM symposium on Operating systems principles (SOSP'97)*, no. October. Saint-Malo, France: ACM, 1997, pp. 129–142. [Online]. Available: http://www.pmg.lcs.mit.edu/papers/iflow-sosp97.pdf

[11] A. C. Myers, "JFlow: Practical Mostly-Static Information Flow Control," in *Proc. of the 26^{th} ACM Symposium on Principles of Programming Languages (POPL'99)*, no. January. San Antonio, Texas, USA: ACM, Jan. 1999. [Online]. Available: http://www.cs.cornell.edu/andru/papers/popl99/popl99.pdf

[12] EASA, "Notification of a Proposal to Issue a Certification Memorandum: Software Aspects of Certification," EASA, Tech. Rep., Feb. 2011.

[13] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security," *IEEE Journal on Selected Areas in Communications*, vol. 21, Jan. 2003.

[14] D. Greve, "Data Flow Logic: Analyzing Information Flow Properties of C Programs," in *Proc. of the 5^{th} Layered Assurance Workshop (LAW'11)*. Orlando, Florida, USA: Rockwell Collins, Research sponsored by Space and Naval Warfare Systems Command Contract N65236-08-D-6805, Dec. 2011. [Online]. Available: http://fm.csl.sri.com/LAW/2011/law2011-paper-greve.pdf

[15] D. Greve and S. Vanderleest, "Data Flow Analysis of a Xen-based Separation Kernel," in *Proc. of the 7^{th} Layered Assurance Workshop (LAW'13)*, no. December. New Orleans, Louisiana, USA: Rockwell Collins, Research sponsored by Space and Naval Warfare Systems Command Contract N66001-12-C-5221, Dec. 2013. [Online]. Available: http://www.acsac.org/2013/workshops/law/files/LAW-2013-Greve.pdf

[16] S. Chong and A. C. Myers, "Decentralized Robustness," in *Proc. of the 19^{th} IEEE Computer Security Foundations Workshop (CSFW'06)*. Washington, D.C, USA: IEEE, Jul. 2006, pp. 242–253. [Online]. Available: http://dl.acm.org/citation.cfm?id=1155681

[17] L. Zheng and A. C. Myers, "End-to-End Availability Policies and Noninterference," in *Proc. of the 18^{th} IEEE Computer Security Foundations Workshop (CSFW'05)*. IEEE, Jun. 2005, pp. 272–286. [Online]. Available: http://www.cs.cornell.edu/andru/papers/avail.pdf

[18] S. Chong, A. C. Myers, K. Vikram, and L. Zheng, *Jif Reference Manual*, http://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html, Feb 2009, jif Version: 3.3.1.

[19] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "sel4: From general purpose to a proof of information flow enforcement," in *Proc. of the IEEE Symposium on Security and Privacy (SP) 2013*, May 2013, pp. 415–429.

[20] F. Verbeek, J. Schmaltz, S. Tverdyshev, O. Havle, H. Blasum, B. Langenstein, W. Stephan, A. Feliachi, Y. Nemouchi, and B. Wolff, "Formal Specification of a Generic Separation Kernel," Apr. 2014. [Online]. Available: http://dx.doi.org/10.5281/zenodo.47299

[21] H. Blasum, O. Havle, S. Tverdyshev, B. Langenstein, W. Stephan, Feliachi, A. Y. Nemouch, B. Wolff, C. Proch, F. Verbeek, and J. Schmaltz, "EURO-MILS: Used Formal Methods," 2015. [Online]. Available: http://www.euromils.eu/downloads/Deliverables/Y2/2015-EM-UsedFormalMethods-WhitePaper-October2015.pdf

[22] Aeronautical Radio Incorporated (ARINC), "Aircraft Data Network Part 5: Network Domain Characteristics and Interconnection," Apr. 2005.

[23] ——, "ARINC 811: Commercial Aircraft Information Security Concepts of Operation and Process Framework," 2005.

[24] Deutsche Kommission Elektrotechnik Elektronik Informationstechnik im DIN und VDE, "Electric signalling systems for railways - Part 104: IT Security Guideline based on IEC 62443," Sep. 2014.

[25] K. Müller, M. Paulitsch, S. Tverdyshev, and H. Blasum, "MILS-Related Information Flow Control in the Avionic Domain: A View on Security-Enhancing Software Architectures," in *Proc. of the 42^{nd} International Conference on Dependable Systems and Networks Workshops (DSN-W)*, Jun. 2012. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6264665

[26] J. Rushby, "Design and Verification of Secure Systems," in *Proc. of the 8^{th} ACM Symposium on Operating Systems Principles*. Pacific Grove, California, USA: ACM, Dec. 1981.

[27] H. R. Nielson, F. Nielson, and X. Li, "Hoare Logic for Disjunctive Information Flow," in *Programming Languages with Applications to Biology and Security - Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*, ser. Lecture Notes in Computer Science, vol. 9465. Springer, 2015, pp. 47–65.

[28] H. R. Nielson and F. Nielson, "Content Dependent Information Flow Control," in *J.Log. Algebraic Methods Program*, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.jlamp.2016.09.005

[29] T. Maciazek, "Content-Based Information Flow Verification for C," MSc thesis, Technical University of Denmark, 2015.

[30] T. Maciazek, H. R. Nielson, and F. Nielson, "Content-Dependent Security Policies in Avionics," in *Proc. of 2^{nd} International Workshop on MILS: Architecture and Assurance for Secure Systems, 2016*, 2016. [Online]. Available: http://mils-workshop.euromils.eu/downloads/hipeac_literature_2016/07-Article.pdf

[31] P. Vasilikos, "Static Analysis for Content Dependent Information Flow Control," MSc thesis, Technical University of Denmark, 2016.