Cif: A Static Decentralized Label Model (DLM) Analyzer to Assure Correct Information Flow in C

Kevin Müller
and Sascha Uhrig
Airbus Group Innovations
Munich, Germany
Email: Kevin.Mueller@airbus.com
Sascha.Uhrig@airbus.com

Michael Paulitsch
Thales Austria GmbH
Vienna, Austria
Email: Michael.Paulitsch@thalesgroup.com

Georg Sigl
Technische Universität München
Munich, Germany
Email: sigl@tum.de

***Abstract*—For safety-critical and security-critical Cyber-Physical Systems in the domains of aviation, transportation, automotive, medical applications and industrial control correct software implementation with a domain-specific level of assurance is mandatory. Particularly in the aviation domain, the evidence of reliable operation demands new technologies to convince authorities of the proper implementation of avionic systems with increasing complexity. Two decades ago, Andrew Myers developed the Decentralized Label Model (DLM) to model and prove correct information flows in applications' source code. Unfortunately, the proposed DLM targets Java applications and is not applicable for today's avionic systems. 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, highly safety-critical avionics are usually implemented in C. Thus, we adjusted the DLM to the programming language C and developed a suitable tool checker, called *Cif*. Apart from proving the correctness of the information flow statically, Cif is also able to create a dependency graph to represent the implemented information flow graphically. Even though this paper focuses on the avionic domain, 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.

Keywords–Security, High-Assurance, Information Flow, Decentralized Label Model

I. INTRODUCTION

In the domain of aviation, software [1] and hardware [2] development has to follow strict development processes and requires certification by aviation authorities. Recently developers of avionics, the electronics on-board of aircrafts, have implemented systems following the concepts of Integrated Modular Avionics (IMA) [3] to reduce costs. For security aspects, there are recent research activities in the topic of Multiple Independent Levels of Security (MILS) [4][5]. Apart from having such architectural approaches to handle the emerging safety and security requirements for mixed-criticality systems, the developers also need to prove the correct implementation of their software applications. For safety, the aviation industry applies various forms of code analysis [6][7][8] in order to evidently ensure correct implementation of requirements. For security, in particular secure information flow, the aviation industry only has limited means available, which are not mandatory yet. Here, the Decentralized Label Model (DLM) [9] that was developed two decades ago, is a promising approach as it is able to prove correct information flows according to a defined flow policy by introducing annotations into the source code. These annotations allow the modeling of the information flow policy directly on source code level avoiding additional translations between model and implementation. In short, DLM extends the type system of a programming language and ensures that the defined information flow policy using label annotations of variables is not violated in the program flow.

DLM is currently available only for Java [10]. Hence, our research challenge is to adapt this model to the C programming language for being able to use it for highly critical avionic applications. We believe annotated source code allowing to check the information flow against the formally proven DLM will help to achieve future security certifications following the framework of Common Criteria with assurance levels beyond EAL 4 [11][12][13] or equivalent avionics security levels. In this paper, we focus our contributions 1) on demonstrating the powerful features of our DLM instantiation for the C language called *Cif*, 2) including the ability of graphically represent the information flows in C programs, and 3) on presenting and discussion common use case examples presented in C code snippets. The instantiation of DLM to C extends its field of use to verify information flow properties to high-assurance embedded systems. The great importance of this research can only be acknowledged if safety software development of aerospace systems and its (in-)ability to use Java has been fully understood. Java is a relatively strongly typed language and, hence, appears at first sight as a very good choice. Among others, the dynamic character of object-oriented languages such as Java introduces additional issues for the certification process [14]. Furthermore, common features such as the Java Runtime Environment introduces potentially unpredictable and harmful delays during execution, which are not acceptable in high-criticality applications requiring high availability and real-time properties like low response times (e.g., avionics).

The remainder of this paper is organized as follow: 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

 ISBN: 978-1-61208-438-1

of the graphical information flow output of our tool. Finally, we conclude our work in Section VI.

II. Related Work

Sabelfeld and Myers present in [15] an extensive survey on research of security typed languages within the last decades. The content of the entire paper provides a good overview to frame the research contribution of our paper. Myers and Liskov present in [9] their ideas of the decentralized trust relation in program information flows called DLM. The authors instantiated DLM to the programming language Java. Known applications (appearing to be of mostly academic nature) using Jif as verification method are:

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

In this paper, DLM is adapted to the programming language C for extending the field of use to high-assurance embedded systems. In [16] Nielson et al. present their research work on the application of DLM and propose improvements to the model that have been identified as useful during the application activities. Both research groups, Nielson's one and the author's one, have been in close exchange in the recent years, particularly discussing the application of DLM to C and discovering related use cases.

Greve proposed in [17] the Data Flow Logic (DFL). This C language extension augments source code and adds security domains to variables. Furthermore, flow contracts between domains can be formulated. 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 [18]. Compared to this approach of using mandatory access control, we used a decentralized approach for assuring correct information flow in this paper. 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.

Khedker et al. published a book [19] on several theoretical and practical aspects of data flow analysis. However, Khedker does not mention DLM as technology. Hence, the DLM research extends his valuable contributions.

III. Decentralized Label Model (DLM)

A. General Model

The DLM [9] is a language-based technology allowing to prove correct information flows within a program. 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. On this data they define read (confidentiality) and write (integrity) policies for other principals in order to grant access to it. Confidentiality policies are expressed by `owners->readers`. Integrity policies use the syntax: `owners<-writers`. 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 provided in [20]. In practice, DLM policies are expressed by *labels* that annotate variables in the source code. 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 braces as *token*. The remainder will use the compiler technology-based term *token* and the DLM-based term *annotation* as synonyms. In the example of Listing 1, 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 the confidentiality or readers policies. In this example, the owner Alice allows Bob to read the data. The second part of the label `Alice<-_` 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 both, the beliefs of Alice and Bob, have to be fulfilled [21].

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 *correct* information flow is allowed if data flows from a source to an *at least equally restricted* destination. 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

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., by observing the status of x it is possible to retrieve the value of y. However, y is more restrictive than x, i.e., x is not allowed by the defined policy to observe the value of y. Thus, the flow in Listing 3 is invalid.

 ISBN: 978-1-61208-438-1

```
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

To analyze those implicit flows, DLM also examines each instruction against the current label of the Program Counter (PC). Using Jif as template, the PC represents the current context in the program and not the actual program counter register [22]. 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 re-calculated at its entrance depending on the condition the block has been entered.

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

A. Extending the C Language with DLM Annotations

1) 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 special 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 worst-case the checker indicates that a program's information flow is correct without verification of labels. Hence, it would introduce the risk of false-positives.

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. This plain C source file can be used as input file for further source code verifications, e.g., by Astrée [7], or as input for the compilation of the final binary.

2) Syntax Extension of C Language: For the format and semantics of annotations, we decided to adapt the concepts of Java Information Flow (Jif) [22], the DLM implementation for Java. We use curly braces 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, pointer overflows reasoned by pointer calculations are not further monitored by Cif. We expect that such coding errors can be covered by additional tools, such as Astrée [7]. 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.

3) 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 access 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 that returns *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.

 ISBN: 978-1-61208-438-1

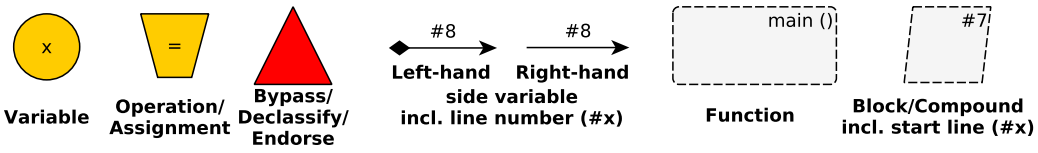

Figure 1. Legend for Flow Graphs

```
int {Alice->Bob} func {param} (int {Alice->*} param) : {Alice->*};
```

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

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 *pseudo-polymorphism*, i.e. it is possible 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 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 full polymorphism, i.e. the inheritance of parameter labels to variable declarations inside the function's body.

V. Use Cases

This section demonstrates the power of Cif by showing some real-world 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 to 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 shows the line of the command's or flow's implementation in the source code.

A. Direct Assignment

The first use case presented in Listing 5 is a sequence of normal assignments.

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

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.

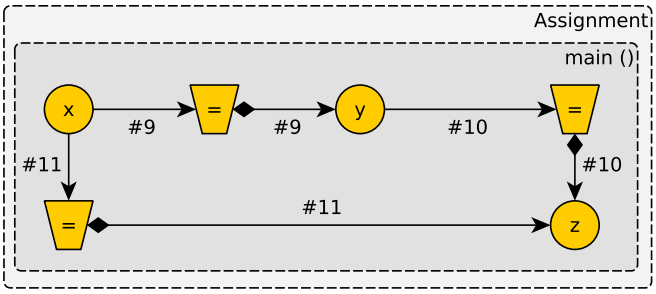

Figure 2. Flow Graph for Listing 5

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 z. However, x and y are less restrictive and, hence, a flow to x in the assignment 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

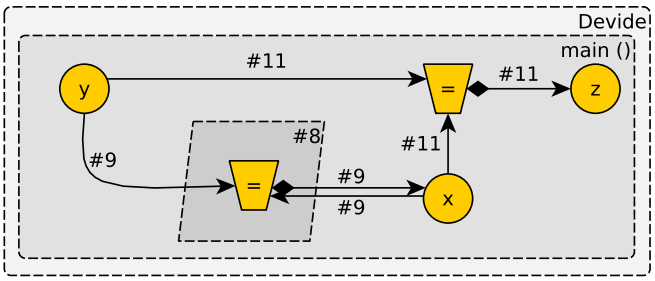

Figure 3. Flow Graph for Listing 7

Remarkable in Figure 3 is the assignment operation in line 9, represented inside the block environment of the if statement but depending on variables outside the block. Hence, Cif parses the code correctly. Also note, that 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

C. Function Calls

A more sophisticated example is the execution of functions. Listing 9 shows a common function call using pseudo-polymorph DLM annotations. The function is called two times with different parameters on line 14 and line 15. 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.

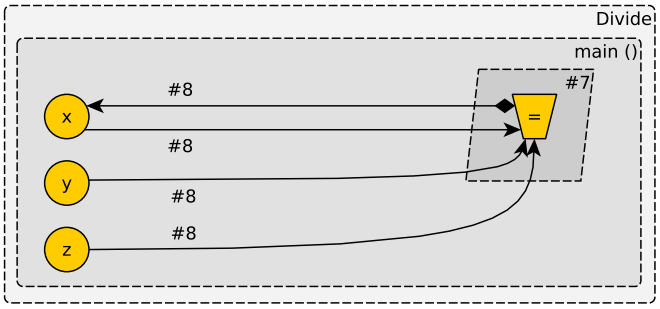

Figure 4. Flow Graph for Listing 8

```
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

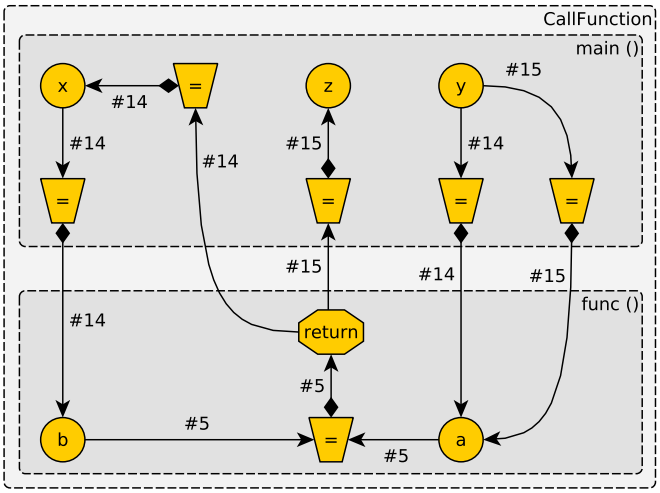

Figure 5. Flow Graph for Listing 9

D. Declassify, Endorse and Bypassing the PC

1) Using Declassify and Endorse: Strictly using DLM forces the developer to model information flows from a low restrictive source to more restrictive destinations. This unavoidably runs into 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 of this flow b is less restrictive in its integrity policy than a, since `Alice` believes that `Bob` is not allowed to 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 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

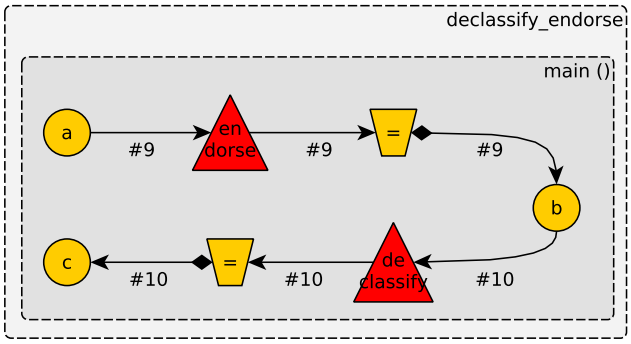

Figure 6. Flow Graph for Listing 10

2) Bypassing the PC label: In 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 $loggedIn$ shall be set to 1 to identify other parts of the application that the user is logged in. This status variable is owned by the principal $System$ and only this principal is allowed to read the variable. The input variables uID and $pass$ are both owned by the principal $User$. 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 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 actual violation of the information flow, 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

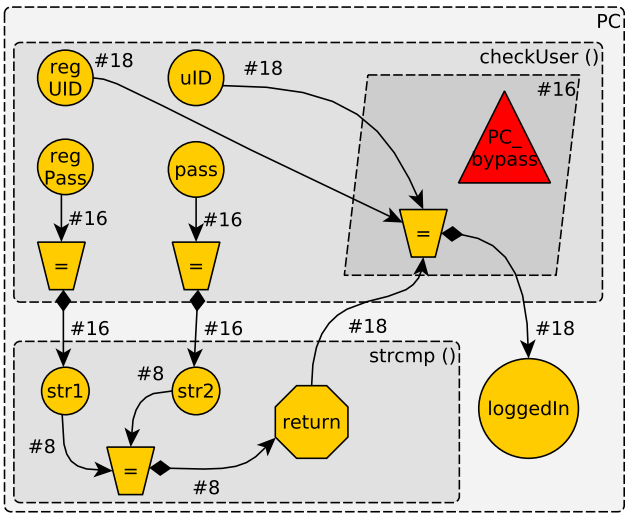

Figure 7. Flow Graph for Listing 11

VI. Conclusion

In this paper 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) [9] for the programming language C. To the best of our knowledge this is the first time DLM is applied to the C language. During the application of DLM to C, we tried to stick to the reference implementation of Java/Jif. However, we had to discuss and 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. We then 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 occuring flows and dependencies. This covers direct assignments of variables, logical and arithmetic operations, indirect dependencies due to decision branches and function calls. DLM also introduces operations to intentionally loosen the strict flow provided by the model. These methods are called endorsement and declassification. Cif also implements these possibilities and specially marks them inside 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.

We also used Cif to implement and verify a larger internal demonstrator project. For high assurance and verification reasons, the demonstrator uses a loosely coupled software design (inspired by [5]) composed of several components with local, analyzable security services working together to provide the software's services. The information flow modeling using annotations helped us to concentrate the implementation on the component's functional purposes only. Furthermore, the information flow evaluation of the component identified several issues in the source code and, finally, could increase the code quality significantly. Particularly, the visualization of indirect flows, e.g., Listing 7 or Listing 8, and function calls, e.g., Listing 9, was very useful during the evaluation. Additionally, this activity showed that Cif is able to cover larger projects, too.

Finally, Cif allows to verify information flows in application implementations with a high level of assurance. This pioneers to create sufficient evidence for security evaluation on high assurance levels, e.g. EAL 7 of the Common Criteria.

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 want to express our gratitude to our project partner at the Danish Technical University, in particular Flemming Nielson. We also thank Kawthar Balti for her input.

References

[1] 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.

[2] ——, "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.

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

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

[5] 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)*. Boston, MA, USA: IEEE, Jun. 2012, pp. 1–6.

[6] 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.

[7] 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.

[8] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[9] 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)*. Saint-Malo, France: ACM, 1997, pp. 129–142.

[10] 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)*. ACM, Jan. 1999, pp. 228–241.

[11] Common Criteria, "Common Criteria for Information Technology Security Evaluation - Part 1: Introduction and general model," 2009.

[12] ——, "Common Criteria for Information Technology Security Evaluation - Part 2: Security functional components," 2009.

[13] ——, "Common Criteria for Information Technology Security Evaluation - Part 3: Security assurance components," 2009.

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

[15] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security," *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 5 – 19, Jan. 2003.

[16] H. R. Nielson, F. Nielson, and X. Li, "Disjunctive Information Flow," DTU Compute, Technical University of Denmark, Denmark, 2014.

[17] 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.

[18] 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)*. New Orleans, Louisiana, USA: Rockwell Collins, Research sponsored by Space and Naval Warfare Systems Command Contract N66001-12-C-5221, Dec. 2013.

[19] U. P. Khedker, A. Sanyal, and B. Karkare, *Data Flow Analysis: Theory and Practice*. CRC Press, 2009.

[20] 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.

[21] 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.

[22] 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; [retrieved: Sept, 2015].

 ISBN: 978-1-61208-438-1