Content-Dependent Security Policies in Avionics

Tomasz Maciążek
tomaciazek@gmail.com

Hanne Riis Nielson
hrni@dtu.dk

Flemming Nielson
fnie@dtu.dk

Department of Applied Mathematics and Computer Science
Technical University of Denmark
Richard Petersens Plads, Building 324
2800 Kongens Lyngby, Denmark

ABSTRACT

We describe a tool (CBIF) for Content-Based Information Flow Control for a subset of the C programming language and show how to use it on applications from the avionics industry. In particular, we consider the *secure gateways* used in the *separation kernels* of operating systems designed according to the principles of Multiple Independent Levels of Security (MILS).

1. INTRODUCTION

We consider programming of Integrated Modular Avionics Systems using the principles of Multiple Independent Levels of Security (MILS) [11]. Here modularity is achieved by means of enforcement of separation and of restriction of communication, using *separation kernels* capable of securely partitioning resources, and *secure gateways* that can examine the content of the messages exchanged and filter them. Müller et al. [5] introduced a detailed specification and architecture of a secure gateway suited for the avionics industry.

There are several requirements that the secure gateway should meet. The most important one is that it should allow *content-based* flow control, that is, it should be possible to examine the content of the messages and not only to determine which partitions should be able to communicate with each other. For parts of those gateways, such as filters, content-based policies and routers are external to the well established and verified secure systems, and thus need to be additionally verified. These parts may consist of relatively small pieces of code that could be verified using static program analysis.

Our examples are motivated by a problem from [4] of routing data to the appropriate transport layer. To be specific, [4] provides the code for a *demultiplexer* module responsible for making the right choice based on the content of the inbound messages and proposes to use the Decentralized Label Model (DLM) [7] for devising policies; subsequently it uses a static analysis to ensure that the program is correct with respect to the information flow security which in this case means that both TCP and UDP packets are correctly forwarded to, respectively, the TCP and UDP modules.

In our view the policies need to be content-dependent in order to provide the required guarantees. A first solution to this is proposed in [10] for a language of concurrent processes and set-based policies. The approach is later refined in [9], featuring more DLM-like policies, and it is the basis of the type system presented in [2], where the CBIF tool implementing the type system is described in detail. An overview of the system is described in [3]. This article heavily draws on [3] but focuses on the amount of guidance provided by the CBIF system to programmers of avionics systems.

```
1 struct s {
2   int {{Alice->Bob,Chuck}} det;
3   int *data;
4 }{
5   (self.det == 1 => self.data={Alice->Bob});
6   (self.det == 2 => self.data={Alice->Chuck})
7 };
8 struct s input;
9
10 int out_chan{
11   (self.index == 0 => self={Alice->Bob});
12   (self.index == 1 => self={Alice->Chuck})
13 } [2];
14 int counter = 0;
15 while(counter < 2)[counter >= 0] {
16   if(input.det == counter + 1) {
17     out_chan[counter] = input.(*data);
18   }
19   counter = counter + 1;
20 }
```

Listing 1: A program with policy annotation

To illustrate the capabilities of the CBIF tool consider listing 1. It is an example of a C program with content-dependent policies that our tool is capable of analysing. The syntax and meaning of the policies will be explained in section 3. The code implements a simple demultiplexer which forwards the given `data` from the `input` structure to the appropriate output channel, which is represented by an array. Paired with the data is a determinant `det` that determines the kind of information and the channel to which it belongs. The `input` structure is not initialized – it is an example of how a program can be modelled. The actual values present in it can be abstracted away and it should be valid for any input. This program is correct with respect to the information flow specified by the policies, and the output of the CBIF tool is:

```
Validation has been completed successfully. There
    ↪ are no illegal flows in the program.
```

In the remainder of this paper we will explain the policies and the validation rules underlying the analytical capabilities of the CBIF tool.

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

Listing 2: Example DLM labeling

2. DECENTRALIZED LABEL MODEL

In this section we review the basic ideas behind DLM – a labelling system for ensuring information flow security. It was originally developed for confidentiality [7] but was later augmented with integrity in [8]. It offers means of declassifying (downgrading confidentiality) and endorsing (downgrading integrity) data and handles implicit information flows in a controlled manner.

DLM is based on a notion of *principals*, that is, entities that can perform some actions in the system and they will act as owners, readers and writers of data; we can think of the principals as a perfect representation of the security domains in avionics. The security policies in DLM take form of *labels*, which consist of owners, as well as readers and/or writers. Labels are associated with data and have the form:

$$\{O_1 \to R_1; ...; O_n \to R_n; O_1 \leftarrow W_1; ...; O_n \leftarrow W_n\}$$

Here O_i is a set representing *owners*, however, for simplicity of policies defined later on, we are going to enforce that it can be only either a singleton (one owner), or the set of all principals (denoted PRIN), or an empty set. Then R_i is a set of *readers* designated by the owners, and W_i is the set of writers who the owners believe that might have influenced the data.

The labels are partially ordered by the $\sqsubseteq$ relation, which is defined as follows:

$$\begin{aligned} L_1 \sqsubseteq L_2 \text{ iff } \forall p : \mathsf{readers}(L_1, p) &\supseteq \mathsf{readers}(L_2, p) \\ \wedge\, \mathsf{writers}(L_1, p) &\subseteq \mathsf{writers}(L_2, p) \end{aligned}$$

where p is a principal, while $\mathsf{readers}(L, p)$ and $\mathsf{writers}(L, p)$ are defined in the following manner:

$$\mathsf{readers}(O \to R, p) = \begin{cases} \{p\} \cup R & \text{if } p \in O \\ \text{PRIN} & \text{otherwise} \end{cases}$$

$$\mathsf{readers}(L_1; L_2, p) = \mathsf{readers}(L_1, p) \cap \mathsf{readers}(L_2, p)$$

$$\mathsf{writers}(O \leftarrow W, p) = \begin{cases} \{p\} \cup W & \text{if } p \in O \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{writers}(L_1; L_2, p) = \mathsf{writers}(L_1, p) \cup \mathsf{writers}(L_2, p)$$

In other words, $\mathsf{readers}(L, p)$ is the set of readers designated by p in label L with p itself included; this means all the readers that p allows to read the data. If p is not an owner in the label it allows reading for all principals by default. The $\mathsf{writers}(L, p)$ is the set of writers designated by p in label L and p itself is included here as well; this means all the writers that p believes may have influenced the data. If p is not an owner in the label it believes by default that no one has influenced the data.

An example of labelling is presented in listing 2. The part of label with the right-arrow (`->`) concerns the confidentiality and the part with the left-arrow (`<-`) concerns integrity. The `*` and `_` signs are the syntax equivalent for the PRIN and $\emptyset$ sets of principals, respectively. In the example, *Bob* is allowed to read the variable `x` by *Alice* and *Alice* is also an implicit reader. *Bob* also believes that no one (but him)

```
int {{Alice->Bob; Bob<-_}} x;
int {(self == 2 => {Alice->_; *<-*})} y;
```

Listing 3: Example content-dependent labelling

has influenced the variable. As for `y`, *Alice* does not allow anyone (but herself) to read it, however, everyone believes that anyone might have influenced the data.

A major advantage of DLM is that it has already been implemented for Java programs as *Jif: Java + information flow*. Jif has originated from JFlow [6] (an early implementation of DLM on a Java-like language), and has developed since – now it also incorporates integrity labelling and many other features that facilitate secure programming and strengthen security. Although it is C, not Java, that is used as the programming language in avionics (due to numerous challenges in assurance), both these languages are imperative, and their principles are similar. Therefore, much of the content-independent part of the solution for C could be based on the Jif documentation [1] and then tested against its implementation. This includes the logic behind the principals, labelling, explicit and implicit flows and other features not discussed in detail in this paper.

3. CONTENT-DEPENDENT POLICIES

In content-dependent policies, the policies are allowed to specify *conditions* on the data and slots (identifiers of places in program's memory), and if these are met then some other policy is applied, which might actually be a *result* that resolves to appropriate labelling (with a DLM *label*). If no condition is specified then it is assumed to be equivalent to `true` and the policy always holds. Note that the slot to which the policy is applied restricts the scope of conditions and results to that slot, and its components (if applicable); actually the slot can be referred to in the condition and in the result as `self`. There can be several results of a policy, for example, concerning different slots. If no slot is specified then by default it is the slot to which the policy is applied that is concerned (i.e. `self`).

An example of a simple policy specification is shown in listing 3. Here, we have an unconditional policy for `x` that effectively is equivalent to the simple DLM label described in section 2. The introduction of policies adds the outer curly braces that delimit the policy specification, while the inner braces remain delimiters of DLM labels. As for `y`, it is governed by a policy that assigns the specified label to it only if its value is equal to 2.

So far we have discussed the syntax of the policies that will be used in the code. However, in order to make use of the policies in the validation process, they need to be processed and combined into a concrete (without `self`) global policy defined as follows, using simplified BNF notation:

$$\begin{aligned} \mathtt{P} ::= {} & X : L \\ \mid {} & \phi \Rightarrow \mathtt{P} \\ \mid {} & \mathtt{P}_1; \mathtt{P}_2 \\ \phi ::= {} & x = n \\ \mid {} & \phi_1 \wedge \phi_2 \\ \mid {} & \phi_1 \vee \phi_2 \end{aligned}$$

The first alternative of the first rule represents assignment of a DLM label L to a set of slots X, initially containing

exactly one slot. The second alternative is a conditional policy, where ϕ is a condition that has to be met for the policy `P` to hold. Furthermore, x is a slot and n is a constant. The use of a set of slots might seem excessive here, but it is necessary for the formal type system that implements the validation.

The translation from the policies appearing in the code to the global policy `P` happens in a *compilation* phase. As a result the policies are concretized with the slots to which they actually apply. For example, the following global policy is derived from the code presented in listing 1:

$$\begin{aligned}\texttt{P} = {} & input.det : Alice \rightarrow Bob, Chuck; \\ & (input.det = 1 \Rightarrow input.data : \{Alice \rightarrow Bob\}); \\ & (input.det = 2 \Rightarrow input.data : \{Alice \rightarrow Chuck\}); \\ & (out\_chan.index = 0 \Rightarrow out\_chan : \{Alice \rightarrow Bob\}); \\ & (out\_chan.index = 1 \Rightarrow out\_chan : \{Alice \rightarrow Chuck\})\end{aligned}$$

4. A FRAGMENT OF C

The language that the CBIF tool is capable of processing is based on a subset of the C language containing:

- Integer, decimal (float) and boolean variables
- Declarations, definitions and assignments
- Arithmetic and boolean operations
- Conditional statements (*if* conditionals)
- Simple iteration statements (*while* loops)
- Structures and structure initialization
- Static and dynamic arrays
- Pointers to simple data types and non-cyclic structures
- Assignments of addresses to pointers
- *malloc* and *sizeof* operations

The language defines two types of slots: variables – instances of simple type data structures; and instances of structures. Following the C standard, structure definitions may contain both variables and structure instances, which are declared with their *short names*. The *fully qualified* names will be those used in all other program instructions and within policies. For example, the structure s from listing 1 has two components `det` and `data`. Once the structure is instantiated in `input` these components can be referenced using their fully qualified names `input.det` and `input.data` respectively for this structure instance.

The language incorporates policies, which can be specified for variables, structures and their instances. The policies are declared inside curly braces when declaring a structure or a variable, after the type specification. More than one policy can be specified using semicolon as a separator. The scope of the policies (the slots on which they are dependent and which they influence) is limited by the point of attachment. Hence, in case of a variable, the scope is that variable itself. As for a structure, it is all its components, and subcomponents if the structure contains nested structures. The programmer may attach the policy to both the structure and its components, so that he has macro- and micro-control over the policies.

Finally, the programmer may specify a *loop invariant* inside brackets before the loop body (as in line 15 of listing 1). It has been introduced in order to avoid fixed-point analysis, which would be otherwise necessary to reason about the state inside loops.

5. PROGRAM VALIDATION

We now give an informal description of the basic content-based program validation. We focus on effects of policies on variables and structures, variable assignments and control statements, leaving out more complex topics, such as structure initialization, structure assignments, pointers and arrays; a discussion of these aspects can be found in [2].

Let us start from defining the context that is provided by the *if* conditionals and *while* loops. These statements create blocks through which not all execution paths are passing, and thus, if an assignment occurs inside those statements, then some information is passed to the assigned slot about the slots that are present in the conditions (the boolean expressions guarding both *if* and *while* statements). In order to register that phenomenon, we will maintain a set X of slots that implicitly influence the current program statement; in other words, the set X will contain all those variables about whose values we might learn something by merely knowing that the given program point was reachable. For both *if* and *while* statements the set X will include all variables present in the condition and the similar sets of variables for the enclosing blocks.

Another information that the context provided by the statement blocks yields are the constraints on the actual values of the variables appearing in those blocks. These constraints also result from the conditions and are useful for determining which policy should apply. The constraints holding at any given statement will be denoted by ϕ_{pc}, which will encompass conditions from all enclosing blocks and results of assignments inside and outside them.

Now, given the knowledge about the context we can define the rule for the assignment. An assignment of form $xv = e$ can only be valid if the policy of the variable, written $\underline{xv}$, is at least as restrictive as the policy of the expression $\underline{e}$ (i.e. $\underline{e} \sqsubseteq \underline{xv}$), where $\underline{e}$ is an aggregation of the policies of variables appearing in the expression e. Furthermore, $\underline{xv}$ must also be at least as restrictive as $\underline{pc}$ (i.e. $\underline{pc} \sqsubseteq \underline{xv}$) – the label of the program counter, which results from joining policies of the slots in X indicating the implicit flow of information.

It is also possible that xv is actually a field of some structure, or such fields are present in e. Then, also the policies of the ancestor structures need to be taken into consideration. Let us assume that $\underline{xv}$ encompasses the policies attached to the variable xv and the policies governing the ancestor structures. The same applies to all variables in e so that $\underline{e}$ is joining their policies.

Finally, the actual labels will be determined by the conditions attached to the policies. The constraint environment ϕ_{pc} holding *before* the assignment statement will determine which labels should be applied to the expression e, while the constraints ψ_{pc} holding *after* the assignment will do the same for xv. We denote this kind of selection of policies by writing the constraint environments in subscripts. The full expression for validating the assignment will then be:

$$\underline{e}_{\phi_{pc}} \sqcup \underline{pc}_{\phi_{pc}} \sqsubseteq \underline{xv}_{\psi_{pc}}$$

The last validated aspect of code are loop invariants, if

```
Validation failed. Offending statement (line 17):
out_chan[counter] = input.data;

Reason:
LHS policy is more restrictive than RHS policy

LHS policy:
((((((counter >= 0) && ((counter:5 == 0) && true)) && (counter < 2)) && (input.det == (counter + 1))) &&
  (true && (false || (out_chan.index == counter)))) =>
input.det|out_chan={Alice->Bob,Chuck};
((input.det == 2) => input.data|out_chan={Alice->Chuck});
((input.det == 1) => input.data|out_chan={Alice->Bob});
((out_chan.index == 1) => ={Alice->Bob});
((out_chan.index == 0) => ={Alice->Bob}))

RHS policy:
input.det={Alice->Bob,Chuck};
((input.det == 2) => input.data={Alice->Chuck});
((input.det == 1) => input.data={Alice->Bob});
((counter == 1) => out_chan={Alice->Bob});
((counter == 0) => out_chan={Alice->Bob})

Model:
out_chan:{Alice} ->[input.det=2, out_chan.index=1, counter=1]
```

Listing 4: Output of the tool for flow validation failure

specified by the programmer. The provided invariant must be true before each execution of the loop, and after it.

Formally, and in the actual implementation (as shown in outputs in section 6), labels are not extracted in the manner presented in this validation specification, which has been given here in order to avoid introducing a complex type system. Instead, the global policy is modified into two versions to reflect the information flow, and these two versions are compared for all possible states to determine whether the flow is valid. The modification is limited to the sub-policies regarding the assigned variable, so that the two resulting policies are comparable against each other on that variable. For example, if we would like to validate the assignment at the end of this program:

```
int {{A->B}} x = 2;
int {(self == 1 => {A->B});
     (self == 2 => {A->B,C})} y;
y = x;
```

Then the following two policies would be created for comparison:

$$\begin{aligned} \mathtt{P}_{left} = (&\mathtt{true} \Rightarrow x, y : \{A \rightarrow B\}); \\ &(y = 1 \Rightarrow \emptyset : \{A \rightarrow B\}); \\ &(y = 2 \Rightarrow \emptyset : \{A \rightarrow B, C\}) \\ \mathtt{P}_{right} = (&\mathtt{true} \Rightarrow x : \{A \rightarrow B\}); \\ &(2 = 1 \Rightarrow y : \{A \rightarrow B\}); \\ &(2 = 2 \Rightarrow y : \{A \rightarrow B, C\}) \end{aligned}$$

The $\mathtt{P}_{left}$ policy represents the fact that something with label $\{A \rightarrow B\}$ is assigned to y, while $\mathtt{P}_{right}$ indicates which policy will be applying to y after the assignment. These two policies are compared on the effective labels assigned to y. In this case the validation would fail since in $\mathtt{P}_{right}$ the y variable gets a less restrictive $\{A \rightarrow B, C\}$ label.

6. THE CBIF TOOL

There are three key technologies that constitute the CBIF solution. First of all, it has been programmed in C# using the .NET 4.5 framework. The ANTLR parser generator has been used to generate code that parses the input and instantiates it as an Abstract Syntax Tree (AST) for further processing. Last but not least, we have taken advantage of a Satisfaction Modulo Theories (SMT) solver, namely Microsoft's Z3, in order to solve the complex, conditional policy comparisons that result from the validation rules.

The validation process starts from parsing the input code and generating AST. Then the resulting tree is compiled in order to extract the concrete global policy. Then, the program is validated, one statement at a time, according to the validation rules. The constraint environments and policy comparison expressions are built deterministically in C#, and then translated to the Z3 domain using its API. In this step, bit vectors are used for purpose of set operations on the domains of slots and principals, while all arithmetic, boolean and comparison operations resulting from the code are directly converted to the respective Z3 types. Finally, the resulting comparison is negated, and then verified by Z3. The negation ensures that if the original comparison is unsatisfiable, then a model revealing the problem will be output.

We have already seen the output of the CBIF tool for the listing 1. In order to show how the tool performs for a program containing information flow problems, we are going to break the example in various ways.

Let us first change the policy in line 12, so that it looks as follows:

```
(self.index == 1 => self={Alice->Bob})
```

This should invalidate the flow, as for $input.det == 2$ the label of `input.data` is $\{Alice \rightarrow Chuck\}$, while now for the index equal to `counter` (which is equal to 1), paired with `input.det` by the *if* conditional, the label of `out_chan` is $\{Alice \rightarrow Bob\}$.

```
1 Validation failed. Offending statement (line 15):
2 while ((counter < 2))[(counter >= 0)] {
3   if ((input.det == (counter + 1))) {
4     out_chan[counter] = input.data;
5   }
6   counter = (counter + 1);
7 }
8
9 Reason:
10 Precondition does not match the postcondition
11
12 Expression:
13 ((counter == (0 - 1)) && true)
14
15 ...does not imply:
16 ((counter >= 0) && ((counter:5 == (0 - 1)) &&
       ↪ true))
17
18 Model:
19 n/a:{n/a} ->[counter=-1]
```

Listing 5: Output of the tool for invariant validation failure

The output of the tool is presented in listing 4, where the policies have been reformatted for better legibility. The first two lines identify the statement which causes the problem. Line 5 gives the reason for which the validation has failed, and the lines that follow provide details. Lines 8 and 9 are the (modified) precondition, where line 9 informs about constraints imposed by usage of subscripts. What follows is the rest of the policy representing the old state and the influencing variables, the *LHS* policy, while the *RHS* policy represents the new state and the assigned variable. The most interesting information, however, comes at line 24, where we have the model for which the policy comparison validation fails. First is the slot, followed by the principal and then the state in the form of mapping from variables to their values. This information allows us to pinpoint the problem in the policy comparison. In line 11 we have that *Alice* designates *Chuck* for `out_chan`, while in line 20 with the matching `counter` it is *Bob* that is designated.

As a second example we shall modify the original example so that the `counter` is initialized with -1, instead of 0. Then we get a validation error presented in listing 5, which indicates that the invariant is not met. Here, again the first lines of the output identify the *source* of the problem. The most useful information for identifying what *is* the problem, is provided by the two expressions in lines 13 and 16, as well as the model, which this time does not specify any particular slot nor principal. The expression from line 13 does not imply the one from line 16, because the latter rules out the possibility of `counter` being equal to -1 (`counter:5` is a fresh variable).

One might wonder why do we need the invariant at all. What if we remove it? Then we get another information flow validation error, this time stating the following (full output not included for brevity):

```
1 Validation failed. Offending statement (line 17):
2 out_chan[counter] = input.data;
3 (...)
4 input.det|out_chan={Alice->Bob,Chuck};
5 (...)
6 Model:
7 out_chan:{Alice} ->[input.det=0,
       ↪ out_chan.index=-1, counter=-1]
```

```
1 policy GatewayHandler =
2 {(self.protocol==6 =>self.func={TCP->_;TCP<-_});
3  (self.protocol==11 =>self.func={UDP->_;UDP<-_})};
4 policy Gateway =
5 {(self.u.protocol==6 =>self={TCP->_;TCP<-_});
6  (self.u.protocol==11 =>self={UDP->_;UDP<-_})};
7 struct DeMuxType {
8   int protocol;
9   int* func;
10 };
11 struct DeMuxType handler {GatewayHandler}[3];
12 struct inputType {
13   struct info {
14     int protocol;
15   } u;
16   int buf[65535];
17 } {Gateway} INPUT;
18 int counter = 0;
19 while(counter < 3) {
20   struct DeMuxType {GatewayHandler} DeMux =
         ↪ handler[counter];
21   if(DeMux.protocol==INPUT.u.protocol) {
22     DeMux.func=INPUT.buf;
23   }
24 }
```

Listing 6: The use case scenario code

As we can see, the states for which the policies are compared include one where `counter` is equal to -1. This is because as `counter` is assigned in the loop, and no invariant is provided, the pre-existing information about the counter is weakened out on entering the loop. The problem here again is the policy of `out_chan`, which receives label $\{Alice \rightarrow Bob, Chuck\}$ of `input.det`, as this variable is present in the *if* condition.

7. AVIONICS EXAMPLES

We have also analysed a use case scenario based on the code of the Receiver Component in Enhanced DLM provided in [4], which is shown in listing 6. The code has been adapted to the policy syntax and language discussed in this work. The adaptation process, among other things, excluded the configuration part, where the `handler` was initialized, which is not necessary for the validation – enough information is provided by the policies. Moreover, the function pointer has been replaced by an integer pointer and the function call by an assignment. From the information flow security point of view the code is equivalent and models the original.

The code performs a similar functionality and envisions similar validation problems as the code from listing 1. The difference is that it concerns both confidentiality and integrity, and re-uses policies declared by name. The verification process for this code completes successfully.

An interesting part here is that we need the assignment to a local slot `DeMux` in line 20, and then use that slot, instead of the `handler` array directly, in the *if* conditional that follows. This is because the type system implemented in the validation tool does not reason about the values of elements of arrays. If the `handler` array was to be used directly in the condition, then such reasoning would be unavoidable. Furthermore, as the policy of the `handler` array does not depend on the index, unlike in listing 1, no loop constraint is necessary here.

It is also possible to model a reverse scenario – a multi-

```
policy GatewayHandler =
{(self.u.prot==6 =>self.buf={TCP->_;TCP<-_});
 (self.u.prot==11 =>self.buf={UDP->_;UDP<-_})};
policy Gateway =
{(self.prot==6 =>self.dataBuf={TCP->_;TCP<-_});
 (self.prot==11 =>self.dataBuf={UDP->_;UDP<-_})};
struct MuxType {
  struct info {
    int prot;
  } u;
  int buf[65535];
};
struct MuxType {GatewayHandler} handler;
struct inputType {
  int prot;
  int dataBuf[65535];
} {Gateway};
struct inputType INPUT[3];
int counter = 0;
while(counter < 3) {
  struct inputType {Gateway} in = INPUT[counter];
  struct MuxType {GatewayHandler} Mux =
      ↪ {{in.prot}};
  Mux.buf = in.dataBuf;
  handler = Mux;
}
```

Listing 7: The multiplexer – the reversed use case scenario code

plexer that receives some data from multiple input channels and modulates them into one output channel. A valid code for that scenario is presented in listing 7. The input channels are modelled as an array, and the output channel handler function is represented as a structure, same as the one being the input for the demultiplexer. Also in this use case scenario the CBIF tool validation succeeds indicating that there are no information flow problems.

The most important part of this code is the translation from the input data structure to the output data structure. This translation needs to be done using structure initialization list – atomically with creation of the structure. However, what is interesting here is that `in.dataBuf` is not part of that initialization. It can be assigned afterwards causing no invalid information flow, because `Mux.buf` has a compatible policy, which does not change due to that assignment. Nevertheless, we would not be able to change the value of the `Mux.prot` that way, as this change would have a side effect of changing the policy of the dependent slot, which is `Mux.buf`. That is why such atomic structure initialization is needed.

8. CONCLUSION

The CBIF tool is capable of processing a large subset of the C language including most common control statements, complex data structures, pointers and arrays. Moreover, if some information flow problems are discovered during analysis the tool is capable of producing a human-readable output that precisely identifies the fault. Finally, thanks to statement-wise analysis separation and use of Z3 the tool has very good performance – for most of the simple programs provided here in listings and used in unit tests the answers were delivered in milliseconds.

One might consider introducing policy inference in the tool – a feature that could increase automation of verification in a way that only minimal modifications to the code are needed. However, it would require creating dependencies between all policy comparisons resulting from the code and checking them all at once. As an effect, the tool would not be able to provide a fine grained information about the problems detected in the input. In fact, this approach might make the analysis infeasible for all but very small programs. Another means of automating the verification process could be introduction of polymorphism, which allows defining methods without specifying the return policy, nor the policies of the arguments. The concept is quite simple, yet powerful, as polymorphic methods can be used in different contexts with different policies.

The final step for automation of verification of safety-critical software could be pushing the policy specification outside of the code. In a MILS-based architecture this could be done by declaring the interfaces of the components against which the software inside those components would be validated. Of course, the input and output of the programs would also have to be well-defined and annotated with policies, and possibly restricted to the IPC channels only. On the security gateway level, the policies of interconnecting components should be checked and only communication that is a restriction should be allowed. The gateway could examine the content of the messages to decide what policy applies on both ends. This way we would ensure that there are no illegal flows due to interface incompatibility, which might happen if the components disagreed about the policy.

Acknowledgements.

We should like to thank Michael Paulitsch and Kevin Müller from Airbus for discussions.

9. REFERENCES

[1] Jif: Java + information flow. `http://www.cs.cornell.edu/jif/`.

[2] Tomasz Maciazek. *Content-Based Information Flow Verification for C*. MSc thesis, Technical University of Denmark, 2015.

[3] Tomasz Maciazek, Hanne Riis Nielson, and Flemming Nielson. Content-Dependent Security Policies for C. (To be submitted for publication), 2015.

[4] Kevin Müller, Ximeng Li, Flemming Nielson, Hanne Riis Nielson, and Georg Sigl. Secure Information Flow Control in Safety-Critical Systems. *Unpublished manuscript*, 2014.

[5] Kevin Müller, Michael Paulitsch, Sergey Tverdyshev, and Holger Blasum. MILS-related information flow control in the avionic domain: A view on security-enhancing software architectures. In *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE, 2012.

[6] Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '99*, pages 228–241, 1999.

[7] Andrew C. Myers and Barbara Liskov. A Decentralized Model for Information Flow Control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, number October, pages 129—-142. ACM, 1997.

[8] Andrew C. Myers and Barbara Liskov. Protecting

privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9, 2000.

[9] Hanne Riis Nielson and Flemming Nielson. Content Dependent Information Flow Control. (Submitted for publication), 2015.

[10] Hanne Riis Nielson, Flemming Nielson, and Ximeng Li. Hoare Logic for Disjunctive Information Flow. To appear in Programming languages with applications to biology and security. Essays dedicated to Pierpaolo Degano for his 65th birthday. In *Lecture Notes in Computer Science*, volume 9465. Springer, 2015.

[11] John Rushby. Separation and Integration in MILS (The MILS Constitution). 2008.