Project: DRA FRONT END FILTER PROJECT

Title: Multi-Level Architectural Model

Ref: DS/FMU/FEF/042 *Issue: Revision* : *2.1* *Date:* 5 June 2016

Status: Approved *Type:* Specification

Keywords:

Author:

Name	*Location*	*Signature*	*Date*
R. D. Arthan	WIN01		

Authorisation for Issue:

Name	*Function*	*Signature*	*Date*
R.B. Jones	HAT Manager		

Abstract: A formulation of a somewhat simplified model of the multi-level SWORD database. (for DRA Front End Filter project RSRE 1C/6130.)

Distribution: HAT FEF File
Simon Wiseman

0 DOCUMENT CONTROL

0.1 Contents List

0.2 Document Cross References

[1] DS/FMU/FEF/003. *Formal Security Policy.* G.M. Prout, ICL Secure Systems, WIN01.

[2] DS/FMU/FEF/039. *Proposal and Quotation for Phase 3.* R.D. Arthan, ICL Secure Systems, WIN01.

[3] DS/FMU/FEF/040. *Multi-level Formal Security Policy.* R.D. Arthan, ICL Secure Systems, WIN01.

[4] DS/FMU/IED/WRK057. *Examples of HOL Type Definitions.* R.D. Arthan, Lemma 1 Ltd., `http://www.lemma-one.com`.

[5] *Security Properties of the SWORD secure DBMS Design.* Simon Wiseman, DRA.

0.3 Changes History

Issue 1.1 (2 February 1994) First draft.

Issue *Revision* : *2.1* **(5 June 2016)** Final approved version.

Issue 2.2 Removed dependency on ICL logo font

0.4 Changes Forecast

None.

1 GENERAL

1.1 Scope

Phases 1 and 2 of the FEF project dealt exclusively with the formulation of the security policy defined in [1]. In that formulation SSQL queries are treated as containing information at a single security classification. In phase 3, the intention is to generalise the policy to cater for queries which are structured objects containing components which may be at several different classifications. This document gives an attempt to give an architectural model in HOL similar to the one in [5] but abstracting and/or simplifying away some of the detail.

It constitutes part of deliverable D17 of work package 7c, as described in section 7 of the Proposal for Phase 3, [2].

1.2 Introduction

An index of the names used in the formal specification may be found in Section 6.

1.3 The Specification

The HOL specification here represents an attempt to capture the key ideas from [5, section 3]. The desire to make this uniform with [1] and [3] has led to some restructuring, principally to use functions from input sequences to output sequences rather than sets of sequences of input-output pairs.

The following differences from [5, section 3] are also noted:

1. For compatibity with the single-level policy treatment of [1] and its generalisation in [3], we do not use the "events" of [5] here.
2. We associate with each output of the system a classification indicating whether or not the output is visible to a client. This accords with the discussion in [5, section 2.3] if not with the formalisation in Z.
3. There is a remark in [5] saying that "if a client can observe an object which contains a reference to a second object then the classification of the second object can [always] be observed". The treatment here of objects within other objects currently reflects this. However, the class at the top node of an object (e.g. in an output stream) is taken here to be an existence indicator for the object (as with input objects in [5]). (See also section 4 below.)

1.4 Setting Up

The following ProofPower instructions set up the new theory *fef042* and set the context for the proof tools. The parent theory is the theory *fef040* which contains some general material about multi-level security policies of the sort we are interested in.

SML

```
open_theory"fef040";
(force_delete_theory "fef042" handle _ => ());
new_theory "fef042";
new_parent "wrk057";
new_parent "fef036";
push_pc"hol";
```

2 POLICY

It seems most convenient to view the treatment of SWORD in [5] in terms of a security policy on the DBMS as seen by the trusted clients. This is like the view taken in the single-level formulation of the policy given in [1]. Thus SWORD receives an input stream comprising classification-query pairs and generates an output stream of multi-level objects.

Internally, SWORD is constructed of an Abstract Machine like that described in section 3 of [5] composed with a filter which sanitises the output from the Abstract Machine. In order to consider properties like the result-labelling property, we need to be able to see this far inside the covers; however, that is not necessary in stating the policy. To do this we use the "generic" policy of [3]. This requires us to define when we consider a pair of inputs or outputs as looking the same at a given class.

The remaining subsections of this section develop the necessary definitions starting with the multi-level objects which are the inputs and outputs to the system.

2.1 Objects

The Z specification in [5] starts (in section 3 of that document) with a formalisation of the classified, structured objects with which the SSQL abstract machine is to deal. The classification and the structure of an object are given there by positing the existence of three functions as follows:

Informal Z

$$objectClass : OBJ \to CLASS$$
$$objectContains : OBJ \to TEXT$$
$$objectRefers : OBJ \to (LOC \nrightarrow OBJ)$$

Here, *objectRefers* is there because objects may have "a number of locations that contain references to other objects". In HOL we will take an object to be a tree, each node of which is labelled with the corresponding classification and text. The locations then become implicit: the objects referred to by an object are just its children in the tree. This is perhaps closer to the discussion in section 2 of [5].

The ProofPower example script [4] introduces a type of labels trees suitable for our purpose. The type is parameterised by the type of the labels. These trees are characterised by a polymorphic constructor function *MkTree*. For example, the instance, $\mathbb{N}$ *TREE* of the type[1], represents trees with numeric labels, and the signature of the corresponding instance *MkTree* has signature:

[1] Note that HOL type constructors such as *TREE* are written after their parameter.

Example HOL

$$MkTree : \mathbb{N} \times \mathbb{N}\ TREE\ LIST \rightarrow \mathbb{N}\ TREE$$

indicating that *MkTree* takes as its argument a pair comprising a number (the label) and a list of trees (the children); *MkTree* maps such an argument to the unique tree with the given label and children. A tree representing an SSQL object is to be labelled with the *objectClass* and *objectContains* values for the object.

We will represent *TEXT* values by HOL strings (although the internal details are not important, so we could also use a type variable). Thus we make the following type abbreviations and definitions:

SML

```
declare_type_abbrev("Text", [], ⌜:STRING⌝);
declare_type_abbrev("Obj", [], ⌜:(Class × Text) TREE⌝);
```

HOL Constant

$$\begin{array}{|ll}
 & \textbf{MkObj} : Class \times Text \times Obj\ LIST \rightarrow Obj; \\
 & \textbf{objectClass} : Obj \rightarrow Class; \\
 & \textbf{objectContains} : Obj \rightarrow Text; \\
 & \textbf{objectRefers} : Obj \rightarrow Obj\ LIST \\
\hline
\forall c\ t\ os \bullet & MkObj\ (c, t, os) = MkTree\ ((c, t), os) \\
\wedge & objectClass\ (MkObj(c, t, os)) = c \\
\wedge & objectContains\ (MkObj(c, t, os)) = t \\
\wedge & objectRefers\ (MkObj(c, t, os)) = os
\end{array}$$

2.2 Requests

A request is a pair comprising a classification and an object representing an SSQL query.

HOL Labelled Product

Req

$$\begin{array}{|ll}
 & \textbf{reqClearance} : Class; \\
 & \textbf{reqSsql} : Obj
\end{array}$$

2.3 Coloured Spectacles

We will eventually require several versions of the "coloured spectacles" which we use to view an object at a given class. The first version is like that of [5, section 3]. It is appropriate to the input queries (and later to the data passed from the Abstract Machine and the Filter).

As in [5] "... two objects will appear identical if they have the same classification and either:

1. the clearance does not dominate this

   or

2. the clearance does dominate this, the contents of the objects are identical, and any objects referred to appear identical."

HOL Constant

$$\mathbf{identicalObj} : Class \rightarrow (Obj \leftrightarrow Obj)$$

$$\begin{array}{ll}
\forall c \bullet & identicalObj\ c = \\
& \{\quad (o_1,\ o_2) \\
& \mid\quad \exists\ c_1\ t_1\ os_1\ c_2\ t_2\ os_2 \bullet \\
& \qquad o_1 = MkObj\ (c_1,\ t_1,\ os_1) \\
& \wedge\quad o_2 = MkObj\ (c_2,\ t_2,\ os_2) \\
& \wedge\quad c_1 = c_2 \\
& \wedge\quad (\quad c\ dominates\ c_1 \\
& \qquad \Rightarrow\quad t_1 = t_2 \\
& \qquad \wedge\quad Length\ os_1 = Length\ os_2 \\
& \qquad \wedge\quad Elems\ (Combine\ os_1\ os_2) \\
& \qquad \subseteq\quad identicalObj\ c)\}
\end{array}$$

Here, the library function *Elems* maps a list into the set of its members (in this case pairs of objects) and *Combine* makes a list of pairs (of objects) out of a pair of lists. Thus we are requiring in the last clause on the right-hand side of the implication that corresponding children of the two objects bear the identical object relation to one another at the given class, c.

We use the same idiom in the lifting of *identicalObj* to lists of objects and histories:

HOL Constant

$$\mathbf{identicalObjs} : Class \rightarrow (Obj\ LIST \leftrightarrow Obj\ LIST)$$

$$\begin{array}{ll}
\forall c \bullet & \\
& identicalObjs\ c = \\
& \{\quad (s_1,\ s_2) \\
& \mid\quad Length\ s_1 = Length\ s_2 \\
& \wedge\quad Elems\ (Combine\ s_1\ s_2) \subseteq identicalObj\ c\}
\end{array}$$

Note that, as in [5], the above comparisons do not admit the possibility that an object may need to be invisible to a client. Again as in [5] this is remedied by using a visibility condition. For our formulation of the policy in section 2.4.1 below it is convenient to have the coloured spectacles with the frames adjusted a little. In particular, with the visibility criteria wired in.

HOL Constant

$$\mathbf{VisibleReq} : Class \rightarrow Req\ \mathbb{P}$$

$$\forall c \bullet \quad VisibleReq\ c = \{\ r \mid c\ dominates\ objectClass(reqSsql\ r)\ \}$$

We now define the "sameness" relations for requests and lists of requests:

HOL Constant

$$\textbf{sameRequest} : Class \rightarrow (Req \leftrightarrow Req)$$

$$\forall c \bullet \quad sameRequest\ c = \{ \quad (r_1, r_2) \mid (reqSsql\ r_1, reqSsql\ r_2) \in identicalObj\ c \}$$

HOL Constant

$$\textbf{sameRequests} : Class \rightarrow (Req\ LIST \leftrightarrow Req\ LIST)$$

$$\forall c \bullet \quad sameRequests\ c = \{ \quad (rs_1, rs_2) \mid let\ rs_3 = rs_1 \upharpoonright VisibleReq\ c\ in\ let\ rs_4 = rs_2 \upharpoonright VisibleReq\ c\ in\ \#rs_3 = \#rs_4 \wedge Elems\ (Combine\ rs_3\ rs_4) \subseteq sameRequest\ c \}$$

An individual output from SWORD will be considered to be invisible to a client for which the corresponding request is invisible. The class of the query will therefore be included as part of each output. We take it that this is the role of the class at the top of the tree (which is the same as for inputs). (This effect is achieved in [5] by considering events each of which comprises an input-output pair; however, this does not generalise the earlier single-level treatment.)

HOL Constant

$$\textbf{VisibleOutput} : Class \rightarrow Obj\ \mathbb{P}$$

$$\forall c \bullet \quad VisibleOutput\ c = \{\ ob \mid c\ dominates\ objectClass\ ob\ \}$$

We consider two outputs which a client is cleared to see to be the same iff. they are actually equal. That is to say, we are assuming that when an output has been passed out of SWORD, the client will have full access to all its components if it can access the object at all. Two lists of outputs look the same to a client if, when we remove outputs which the client is not cleared to see, we end up with equal lists of outputs each pair of which are the same in the above sense.

HOL Constant

$$\textbf{sameOutputs} : Class \rightarrow (Obj\ LIST \leftrightarrow Obj\ LIST)$$

$$\forall c \bullet \quad sameOutputs\ c = \{ \quad (os_1, os_2) \mid os_1 \upharpoonright VisibleOutput\ c = os_2 \upharpoonright VisibleOutput\ c \}$$

2.4 The Type of Systems

We consider the SWORD system to be given by its behavioural model which is a function from lists of requests to lists of outputs:

SML

$$declare\_type\_abbrev(\texttt{"}ML\_BEHAVIOUR\texttt{"},\ [],\ \ulcorner{:}Req\ LIST \rightarrow Obj\ LIST\urcorner);$$

2.4.1 Non-interference Policy

The relevant notion of non-interference is then obtained by instantiating the "generic security policy" from [3].

HOL Constant

$$\mathbf{SWORD\_ml\_secure} : ML\_BEHAVIOUR\ \mathbb{P}$$

$$SWORD\_ml\_secure = ml\_secure\ sameRequests\ sameOutputs$$

3 SWORD SYSTEM CONSTRUCTION

3.1 The Multi-level SSQL Abstract Machine

3.1.1 Machines

We do not want to settle on the representation of states here, so we use a type variable $'State$ for these.

HOL Labelled Product

Machine

$$\begin{array}{ll} \mathbf{Next} & : 'State \times Req \rightarrow 'State; \\ \mathbf{Output} & : 'State \times Req \rightarrow Obj; \\ \mathbf{Init} & : 'State \end{array}$$

3.1.2 Histories

The following definition gives a function which transforms a state machine which takes single *Req*s and yields single *Obj*s into one which takes a list of *Req*s and yields a list of *Obj*s.

HOL Constant

lift_machine : $'State\ Machine \to 'State \times Req\ LIST \to Obj\ LIST \times 'State$

$\forall mch\ s\ r\ rl \bullet$

$lift\_machine\ mch\ (s, []) = ([], s)$

$\wedge \quad lift\_machine\ mch\ (s, Cons\ r\ rl) =$

$let\ out = Output\ mch\ (s, r)$

$and\ s' = Next\ mch\ (s, r)$

$in \quad let \quad (outl, final\_state) = lift\_machine\ mch\ (s', rl)$

$\quad\ in \quad (Cons\ out\ outl, final\_state)$

The behavioural model of the machine is then given by:

HOL Constant

Behaviours : $'State\ Machine \to Req\ LIST \to Obj\ LIST$

$\forall mch\ rs \bullet$

$Behaviours\ mch\ rs = Fst\ (lift\_machine\ mch\ (Init\ mch, rs))$

3.1.3 Filter

The filter takes an object and a classification and sanitises the object for a client at that clearance.

HOL Constant

FilterObj : $Class \to Obj \to Obj$

$\forall c\ d\ t\ os \bullet$

$FilterObj\ c\ (MkObj(d, t, os)) =$

$if \quad c\ dominates\ d$

$then \quad MkObj(d, t, Map\ (FilterObj\ c)\ os)$

$else \quad MkObj(d, Arbitrary, Arbitrary)$

3.2 System Construction

We construct a system from the Abstract Machine and the Filter by composing the output function of the machine with the filter and then forming the behavioural abstraction of the resulting machine.

HOL Constant

SWORD_construction : *'State Machine → ML_BEHAVIOUR*

$$\forall mch \bullet\ SWORD\_construction\ mch =$$
$$\quad let\quad sec\_output =$$
$$\quad\quad \lambda(st,\ r) \bullet\ FilterObj\ (reqClearance\ r)\ (Output\ mch\ (st,\ r))$$
$$\quad in\ let\quad sec\_mch = MkMachine\ (Next\ mch)\ sec\_output\ (Init\ mch)$$
$$\quad in\quad Behaviours\ sec\_mch$$

We now want to find conditions on the Abstract Machine which will ensure that the constructed system satisfies the policy of section 2.4.1. To do this in a natural way we need a formulation of the coloured spectacles appropriate to examining the data which passes from the Abstract Machine to the Filter.

HOL Constant

sameFilterInputs : $Class \rightarrow (Obj\ LIST \leftrightarrow Obj\ LIST)$

$$\forall c \bullet$$
$$\quad sameFilterInputs\ c =$$
$$\quad \{\quad (os_1,\ os_2)$$
$$\quad |\quad let\quad os_3 = os_1 \upharpoonright VisibleOutput\ c$$
$$\quad\quad in\ let\quad os_4 = os_2 \upharpoonright VisibleOutput\ c$$
$$\quad\quad in\quad \#os_3 = \#os_4$$
$$\quad\quad \wedge\quad Elems\ (Combine\ os_3\ os_4) \subseteq identicalObj\ c\}$$

HOL Constant

FlowSecureMachine : *'State Machine* $\mathbb{P}$

$$FlowSecureMachine\ =$$
$$\{\quad mch$$
$$|\quad Behaviours\ mch$$
$$\in\quad ml\_secure\ sameRequests\ sameFilterInputs\}$$

SML

$$val\ \textbf{conj\_042\_1} = \ulcorner$$
$$\forall mch \bullet\ mch \in FlowSecureMachine \Rightarrow SWORD\_construction\ mch \in SWORD\_ml\_secure$$
$$\urcorner;$$

4 REMARKS

It is not immediately clear how much further one can usefully go without trying to reconcile the informal description of multi-level results in section 2.3 of [5] with the *identicalObjs* function of section 3 of that document.

It would certainly be possible to carry on the general trend of the above material by trying to strengthen the *sameOutputs* function to make it more accurately reflect the multi-level nature of an output. However, there are probably many choices about how to do this and the informal description of the four levels of the outputs does not really accord with *identicalObjs*.

In the description of the result-labelling property, what happens is, apparently, that one flattens out levels 2 and 3 of a tree to produce a list (or set?) of labelled components and then worries about information flows into an element of the result. How to formalise this is unclear to me (particularly in the light of the discussion of down-grading at the end of section 2.3 of [5]).

5 CLOSING DOWN

The following ProofPower instruction restores the previous proof context.

```
SML
pop_pc();
```

6 INDEX