Detecting Cognitive Causes of Confidentiality Leaks[★]

R. Rukšėnas[a,1], P. Curzon[a], A. Blandford[b]

[a] Dept. of Computer Science, Queen Mary, University of London, London, UK

[b] University College London Interaction Centre, London, UK

Abstract

Most security research focuses on the technical aspects of systems. We consider security from a user-centred point of view. We focus on cognitive processes that influence security of information flow from the user to the computer system. For this, we extend our framework developed for the verification of usability properties. Finally, we consider small examples to illustrate the ideas and approach, and show how some confidentiality leaks, caused by a combination of an inappropriate design and certain aspects of human cognition, can be detected within our framework.

Key words: human error, security, cognitive architecture, formal verification, SAL.

1 Introduction

There has been much research on security (confidentiality) of information flow (see Sabelfeld and Myers' overview [16]). The starting point is the assumption that computation uses confidential inputs. The goal is to ensure a *noninterference policy* [8], which essentially means that no difference in outputs can be observed between two computations that are different only in their confidential inputs. Various approaches to this problem, such as access and static information-flow control [16], have been proposed, and formalisms and mechanisms developed, e.g. security-type systems and type-checkers [16].

All this research focuses on the technical aspects of software systems. It aims at ensuring that the implementation of a system does not leak confidential information. However, technology is only one aspect of security. Within interactive systems, there is another actor besides a computer system – its

[★] This research is funded by EPSRC grants GR/S67494/01 and GR/S67500/01.

[1] Email: `rimvydas@dcs.qmul.ac.uk`

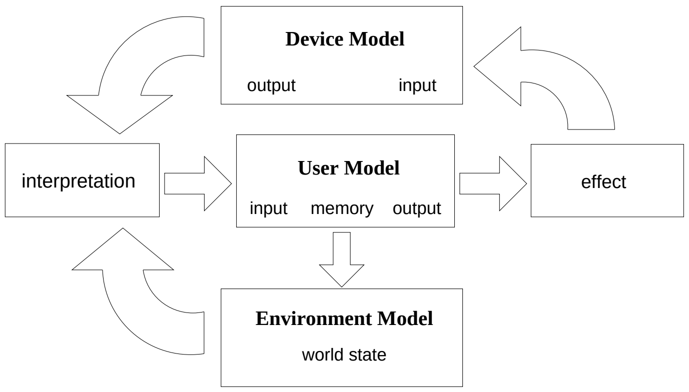

Fig. 1. The cycle of interaction

human user. Even perfectly designed and implemented systems cannot prevent users from unwittingly compromising confidential information they have. Users can breach security for many reasons. Nevertheless, research in human-computer interaction [1,11] reveals systematic causes of such violations, including cognitive overload, lack of security knowledge, and mismatches between the behaviour of computer systems and the mental model that their users have. Even in the absence of software errors security can be breached when the functionally correct behaviour is inconsistent with user expectations [11].

The relationship between users and security mechanisms is addressed by *user-centred security* which provides "security models, mechanisms, systems, and software that have usability as a primary motivation or goal" [17]. Much of this work takes social dimensions, considering problems like user motivation and understanding of security mechanisms, work practices, the relationships between system users, including authorities and communities of users, and threats to security exploiting social engineering techniques.

Our work lies between the technical aspects of information-flow security and the social aspects of user-centred security. More specifically, we are interested in information flow; however, the locus of this flow is now not within a computer system but within the inputs provided to it by its user. We are not considering the social aspects of human-computer interaction and security. Instead, the focus of our attention is cognitive processes that influence information flow from the human user into the computer system.

We build upon the generic user model (cognitive architecture) we developed in our work on usability [7]. It was developed from abstract cognitive principles, such as a user entering an interaction with knowledge of the task and its subsidiary goals. The cognitive architecture was later extended [14] to include an abstract specification, *interpretation*, of the pathways from device signals and environment objects to the user decision of what they mean (see Fig. 1). Incorporating such models of user behaviour into models of security is advocated by user-centred security [17]: e.g. Ka-Ping derives the guidelines (design rules) for secure interaction design from an informal user model [11].

Our cognitive architecture has proved of use for detecting various types of systematic user errors in the context of usability and task completion [7,14]. Here our aim is to show that the behaviours emerging from this architecture also expose security problems and so facilitate the improvement of security aspects in user interaction design. To demonstrate this, we first informally discuss, from a security viewpoint, several examples of user error dealt with in our earlier work [7]. Then we consider an example of using the model checking tool SAL [12] to detect some confidentiality leaks emerging from our cognitive architecture and conditioned by the user interpretation of system prompts. More specifically, we consider security problems that may arise from the combination of user habits and (relative) positioning of input fields in authentication interfaces. The examples are small and intended to illustrate an approach and ideas that we believe are more generally applicable.

Summarising, the main contribution of this paper is the following:

- An investigation into the formal modelling of cognitive aspects of confidentiality leaks in interactive systems.
- An extension of our framework, developed for usability verification, to deal with the security problems in user interaction.
- An illustrative example of confidentiality leaks, caused by cognitive interpretation and detectable by model checking using our cognitive architecture.

Related work

Whilst conducted independently and in parallel, Beckert and Beuster's work [2] takes a similar approach to ours. They also develop a formal user model, and combine it with specifications of the application and the user's assumptions about that application to verify security properties of interactive systems. Their user modelling is based on the formalisation of an established methodology, GOMS [10], which is the core of their work. The modelling of user's assumptions partially coincides with our user interpretation. However, their ”assumptions” model user choice between multiple plausible options, whereas our ”interpretation” deals, in addition, with the user perception of interface objects depending on their shape, position, etc. Beckert and Beuster informally define three HCI security requirements, however, only one is formalised, whereas correctness properties in our framework also address the remaining two. It is also unclear whether they provide tool support for automatic verification. On the other hand, their methodology supports hierarchical models: an advantage when dealing with larger systems.

In the related area of safety-critical systems, Rushby *et al* [15] focus on mode errors and the ability of pilots to track mode changes. They formalise plausible mental models of systems and analyse them using the Murϕ verification tool. The mental models though are essentially abstracted system models; they do not rely upon structure provided by cognitive principles. Neither do they model user interpretation. Cerone *et al*'s [6] CSP model of an air traffic control system includes controller behaviour. A model checker was used

Table 1
A fragment of the SAL language

`x:T`	`x` has type `T`
`λ(x):e`	a function of `x` with the value `e`
`x′ = e`	an update: the new value of `x` is that of the expression `e`
`{x:T|p(x)}`	a subset of `T` such that the predicate `p(x)` holds
`a[i]`	the `i`-th element of the array `a`
`r.x`	the field `x` of the record `r`
`r WITH .x := e`	the record `r` with the field `x` replaced by the value of `e`
`g` → upd	if `g` is true then update according to `upd`
`c [] d`	non-deterministic choice between `c` and `d`
`[](i:T):` $\texttt{c}_\texttt{i}$	non-deterministic choice between the $\texttt{c}_\texttt{i}$ with `i` in range `T`

to look for new behavioural patterns, missed by the analysis of experimental data. The classification stage in their model is similar to user interpretation.

Ka-Ping [11] gives a list of design rules, justified by an informal user model and tailored to increase security of interactive systems. As the rules are informal (many are probably too abstract to be formalised), there is no tool support for verifying whether designs obey them.

2 The Cognitive Architecture in SAL

Our cognitive architecture is a higher-order logic formalisation of abstract principles of cognition and specifies cognitively plausible behaviour [4]. The architecture specifies possible user behaviour (traces of actions) that can be justified in terms of specific results from the cognitive sciences. Real users can act outside this behaviour, about which the architecture says nothing. Its predictive power is bounded by the situations where people act according to the principles specified. The architecture allows one to investigate what happens if a person acts in such plausible ways. The behaviour defined is neither "correct" or "incorrect". It could be either depending on the environment and task in question. We do not attempt to model the underlying neural architecture nor the higher level cognitive architecture such as information processing. Instead our model is an abstract specification, intended for ease of reasoning.

We rely upon cognitive principles that give a *knowledge level* description in the terms of Newell [13]. Their focus is on the goals and knowledge of a user. Our formalisation of the principles is based on the SAL model checking environment [12]. It provides a higher-order specification language and tools for analysing state machines specified as parametrised modules and composed either synchronously or asynchronously. The SAL notation we use is given in Table 1. We also use the usual notation for the conjunction, disjunction and set membership operators. The SAL specification of a transition relation that defines our user model is given in Fig. 2. Below we discuss the cognitive principles and the way they are reflected in the specification (module `User`).

```
TRANSITION
  [](i:GoalRange):   GoalCommit:
     gcommit[i] = ready  ∧
     NOT(gcomm ∨ rcomm)  ∧            gcommit′[i] = committed;
     finished = notf  ∧          →    gcomm′ = TRUE
     goals[i].grd(in, mem, env)
[]
  [](i:ReactRange):   ReactCommit:
     rcommit[i] = ready  ∧
     NOT(gcomm ∨ rcomm)  ∧            rcommit′[i] = committed;
     finished = notf  ∧          →    rcomm′ = TRUE
     react[i].grd(in, mem, env)
[]
  [](i:GoalRange):   GoalTrans:
                                    gcommit′[i] = done;
     gcommit[i] = committed    →    gcomm′ = FALSE
                                    GoalTransition(i)
[]
  [](i:ReactRange):   ReactTrans:
                                    rcommit′[i] = ready;
     rcommit[i] = committed    →    rcomm′ = FALSE
                                    ReactTransition(i)
[]
  Exit:
     PerceivedGoal(in, mem)  ∧
     NOT(gcomm ∨ rcomm)  ∧          →    finished′ = ok
     finished = notf
[]
  Abort:
     NOT(EnabledGoals(in, mem, env))  ∧
     NOT(EnabledReact(in, mem, env))  ∧        finished′ = IF Wait(in, mem)
     NOT(PerceivedGoal(in, mem))  ∧       →                THEN notf
     NOT(gcomm ∨ rcomm)  ∧                                 ELSE abort ENDIF
     finished = notf
[]
  Idle:
     finished = notf    →
```

Fig. 2. User model in SAL

Non-determinism. In any situation, any one of several cognitively plausible behaviours might be taken. It cannot be assumed that any specific plausible behaviour will be the one that a person will follow. The SAL specification is a transition system. Non-determinism is represented by the non-deterministic choice, `[]`, between the named guarded commands (i.e. transitions). For example, *GoalCommit* in Fig. 2 is the name of a family of transitions indexed by `i`. Each guarded command in the specification describes an action that a user *could* plausibly make.

Mental versus physical actions. A user commits to taking an action in a way that cannot be revoked after a certain point. Once a signal has been sent from the brain to the motor system to take an action, it cannot be stopped even if the person becomes aware that it is wrong before the action is taken. Therefore, we model both *physical* and *mental* actions. Each physical action modelled is associated with an internal mental action that commits to taking it. In the specification, this is reflected by the pairings of guarded commands: *GoalCommit* – *GoalTrans*, and *ReactCommit* – *ReactTrans*. The first of the pair models committing to an action, the second actually doing it (see below).

User goals. A user enters an interaction with knowledge of the task and, in particular, task dependent sub-goals that must be discharged. These sub-goals might concern information that must be communicated to the device or items (such as bank cards) that must be inserted into the device. Given the opportunity, people may attempt to discharge such goals, even when the device is prompting for a different action. We model such knowledge as user goals which represent a pre-determined partial plan that has arisen from knowledge of the task in hand, independent of the environment in which that task is performed. No fixed order is assumed over how user goals will be discharged.

To see how this is modelled in SAL consider the guarded command *GoalTrans* for doing a user action that has been previously committed to:

$$\texttt{gcommit}[\texttt{i}] = \texttt{committed} \quad \rightarrow \quad \begin{array}{l} \texttt{gcommit}'[\texttt{i}] = \texttt{done}; \\ \texttt{gcomm}' = \texttt{FALSE}; \\ \mathit{GoalTransition}(i) \end{array}$$

The left-hand side of → is the guard of this command. It says that the rule will only activate if the associated action has already been committed to, as indicated by the `i`-th element of the local variable array `gcommit` holding value `committed`. If the rule is then non-deterministically chosen to fire, this value is changed to `done` and the boolean variable `gcomm` is set to false to indicate there are now no commitments to physical actions outstanding and the user model can select another goal. $GoalTransition(i)$ represents the state updates associated with this particular action i.

User goals are modelled as an array, `goals`, which is a parameter of the `User` module. The state space of the user model consists of three parts: input variable `in`, output variable `out`, and local variable (memory) `mem`; the environment is modelled by a global variable, `env`. All of these are specified using type variables and are instantiated for each concrete interactive system. Each goal is specified by a record with the fields `grd`, `tout`, `tmem` and `tenv`. The `grd` field is discussed below. The remaining fields are relations from old to new states that describe how two components of the user model state (outputs `out` and memory `mem`) and environment `env` are updated by discharging this goal. These relations, provided when the generic user model is instantiated, are used to specify $GoalTransition(i)$ as follows:

$\texttt{out}' \in \{\texttt{x:Out | goals[i].tout(in,out,mem)(x)}\};$
$\texttt{mem}' \in \{\texttt{x:Memory | goals[i].tmem(in,mem,out}'\texttt{)(x)}\};$
$\texttt{env}' \in \{\texttt{x:Env | goals[i].tenv(in,mem,env)(x)} \wedge \textit{possessions}\}$

Since we are modelling the cognitive aspects of user actions, all three updates depend on the initial values of inputs (perceptions) and memory. In addition, each update depends on the old value of the component updated. The memory update also depends on the new value (`out′`) of the outputs, since we usually assume the user remembers the actions just taken. The update of `env` must also satisfy a generic relation, *possessions*. It specifies universal physical constraints on possessions and their value, linking the events of taking and giving up a possession item with the corresponding increase or decrease in the number (counter) of items possessed. For example, it specifies that if an item is not given up then the user still has it. The counters of possession items are modelled as environment components. We omit further details since possession properties are not used in any way in this paper.

If the guarded command for *committing* to a user goal, *GoalCommit*, fires, it switches the commit flag for goal `i` to `committed` thus enabling the *GoalTrans* command. The predicate `grd`, extracted from the `goals` parameter, specifies when there are opportunities to discharge this user goal. Because we assign `done` to the corresponding element of the array `gcommit` in the *GoalTrans* command, once fired the command below will not execute again. If the user model discharges a goal, without an additional reason such as a prompt, it will not do so again.

Reactive behaviour. Users may react to an external stimulus, doing the action suggested by the stimulus. For example, if a flashing light comes on a user might, if the light is noticed, react by inserting coins in an adjacent slot. Reactive actions are modelled by the pairing *ReactCommit* – *ReactTrans* in the same way as user goals but on different variables, e.g. parameter `react` of the `User` module rather than `goals`. $ReactTransition(i)$ is specified in the same way as $GoalTransition(i)$. The array element `rcommit[i]` is reassigned `ready` rather than `done`, once action `i` has been executed, since reactive actions, if prompted, *may* be repeated.

Goal based task completion. Users intermittently, but persistently, terminate interactions as soon as their main goal has been achieved [5], even if subsidiary tasks generated in achieving the main goal have not been completed. A cash-point example is a person walking away with the cash but leaving the card. In the SAL specification, a condition that the user perceives as the main goal of the interaction is represented by a parameter `PerceivedGoal` of the `User` module. Goal based completion is then modelled as the guarded command *Exit*, which simply states that, once the predicate `PerceivedGoal` becomes true and there are no commitments to user goals and/or reactive actions, the user may complete the interaction. This action may still not be taken because the choice between enabled guarded commands is non-deterministic. Task completion is modelled by setting the local variable `finished` to `ok`, whereas the value `notf` means that the task is unfinished.

No-option based task termination. If there is no apparent action that a person can take that will help complete the task then the person may terminate the interaction. For example, if, on a ticket machine, the user wishes to buy a weekly season ticket, but the options presented include nothing about season tickets, then the person might give up, assuming the goal is not achievable.

In the guarded command *Abort*, the no-option condition is expressed as the negation of predicates `EnabledGoals` and `EnabledReact`. Note that, in such a case, a possible action that a person could take is to wait. However, they will only do so given some cognitively plausible reason such as a displayed "please wait" message. The waiting conditions are represented in the specification by predicate parameter `Wait`. If `Wait` is false, `finished` is set to `abort` to model a user giving up and terminating the task.

3 Verification of Security Aspects in User Interaction

In this section, we discuss examples of user error, focussing on the security aspects of interaction. We first introduce the properties to verify.

3.1 Correctness properties: usability and security

Previously, our approach dealt with two kinds of usability properties. First, we want to be sure that, in any possible system behaviour, the user's main goal of interaction (as they perceive it) is eventually achieved. Given our model's state space, this is written as the SAL assertion

(1) $$\mathtt{F}(\mathtt{PerceivedGoal}(\mathtt{in},\mathtt{mem}))$$

where `F` means 'eventually'. Second, in achieving a goal, subsidiary tasks are often generated that the user must complete to complete the task associated with their main goal. If the completion of the subsidiary tasks is represented as a predicate, `SecondaryGoal`, the required condition is (where `G` means 'always'):

(2) $$\mathtt{G}(\mathtt{PerceivedGoal}(\mathtt{in},\mathtt{mem}) \Rightarrow \mathtt{F}(\mathtt{SecondaryGoal}(\mathtt{in},\mathtt{mem},\mathtt{env})))$$

This states that the secondary goal is always eventually achieved once the perceived goal has been. Often secondary goals can be expressed as interaction invariants [7] which state that some property of the system state, that was perturbed to achieve the main goal, is restored. Previously, we viewed property (2) in terms of pure usability, applying it to, e.g. user possessions.

The verification of (2) can, however, also be used to detect security problems. Moreover, we will introduce a third kind of correctness property, relevant to confidentiality leaks in user input. Intuitively, one would like to prevent such leaks in all system states, so we are aiming at a safety property. In terms of information-flow security [16], let us have, for simplicity, two confidentiality levels of user inputs, "high" and "low". A safety property that addresses some security aspects is that in no states do high inputs appear on a low channel. A boolean, `SecurityBreach`, represents system states that breach this. The property, stating that it is always true there is no security breach, is then:

$$\mathtt{G(NOT(SecurityBreach))} \tag{3}$$

We discuss how `SecurityBreach` is set to true, indicating breaches, in Section 4.

Note that neither of the first two correctness properties capture confidentiality leaks modelled as `SecurityBreach`. Property (1) is about usability; the essential condition is a user achieving the main goal. The fact that this goal might be achieved by first making a mistake then undoing the erroneous action is irrelevant. However, with respect to security, undo is not good enough [11]: an erroneous action could already have leaked confidential information. Though checking property (2) can reveal some security problems related to, e.g. post-completion errors (see below), it is still a liveness property. As such, it does not require a system to satisfy the condition `SecondaryGoal` in all states, only at some point after the main goal has been achieved.

3.2 User error and security

Erroneous actions are the proximate cause of failure, since it was a particular action that caused the problem: e.g. a user entering data in the wrong field. To eliminate the problem, however, one must consider the ultimate causes of an error. In our framework, we consider situations where the ultimate causes are aspects (limitations) of human cognition that have not been addressed in the interface. An example is that a person enters data in a particular field because the interface design suggests it as appropriate for that data. In Hollnagel's terms [9] which distinguish between human error *phenotypes* (classes of erroneous actions) and *genotypes* (the underlying, e.g. psychological, cause), our cognitive architecture deals with genotypes. Since there is no evidence that security errors are conditioned by different cognitive causes to usability errors, our cognitive architecture can exhibit behaviours leading to security problems, even though it was developed without security concerns in mind. Some of these errors have the same cognitive causes as the usability errors we dealt with in our earlier work [7]. Next we discuss several types of user error, related to security but still detectable within the usability based approach represented by properties (1) and (2).

A persistent user error that emerges from the cognitive architecture is the post-completion error [5], where a user terminates an interaction with completion of subsidiary tasks outstanding. People have been found to make such errors even in lab conditions [5]. An example of this error, which is also a security breach, is when, with old cash machines, users persistently took cash but left their bank card. Within our cognitive architecture, such behaviour emerges because of an action (*Exit*) that allows a user to stop once the goal has been achieved. Using our verification framework, this is detected by checking

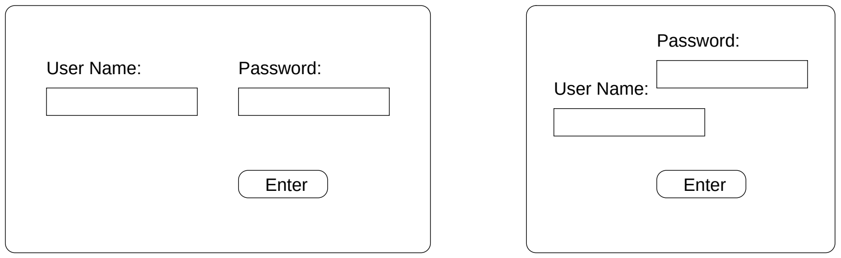

Fig. 3. Two layouts of an authentication interface

property (2). For this, `SecondaryGoal` would state that the total value of user possessions (bank cards included) in a state is the same as it was before the interaction. The formal verification of a similar example is described in [7].

Blandford and Rugg [3] give an example of an extant security breach caused by users forgetting to log out when moving away from an industrial printer, leaving it vulnerable to sabotage – e.g. by unauthorised users changing the printed message, etc. Being a case of the post-completion error, it can be detected by verifying property (2) with the appropriately chosen `SecondaryGoal`.

Previously [14] we also considered user error due to the shape-induced confusion over the meaning of interface prompts. The example was that of a user attempting to top-up a phone card using an ATM. We showed how model checking, based on our cognitive architecture, can identify user confusion as to which of two numbers, phone number or top-up card number, is requested. The property checked was of type (1), i.e. whether the user achieves the main goal. User confusion in a similar situation can also result in confidentiality leaks. For example, asked to enter a card number, a person might be confused whether the number requested is that of a bank card or a phone card. If a bank card number is entered when the interface prompts for a top-up card number, the input might be displayed which is a security breach. This problem would also be detected by analysing why the user could not achieve the main goal.

4 A Case Study: Authentication Interface

In this section, we extend our previous work and investigate how other security problems, not considered in that work, can be detected using our cognitive architecture formalised in SAL. In particular, we show how user habits in combination with some designs, can lead to the incorrect interpretation of interface prompts, resulting in the leakage of confidential information. To determine whether such leakages are possible, we introduce into our framework a new entity, generic module `tester`. This module is instantiated by providing a collection of channels and a high security value. The instantiated module then checks whether this value can appear on one of the low security channels.

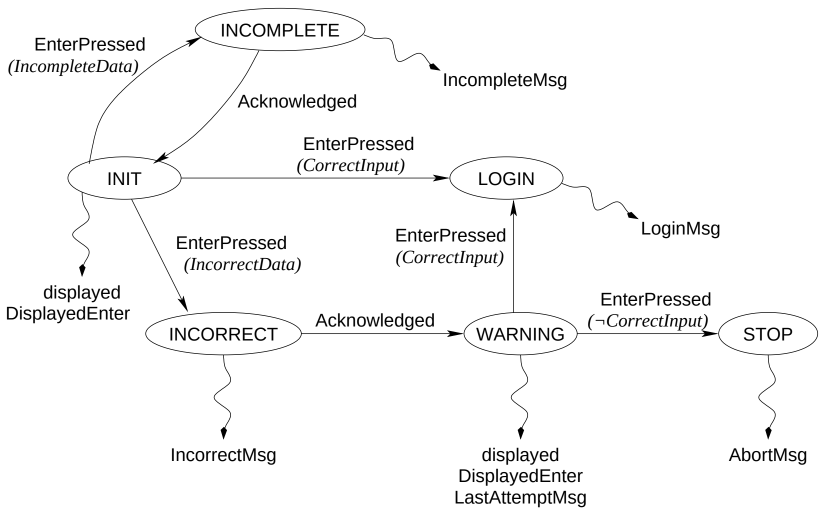

Fig. 4. Authentication procedure

4.1 An Authentication Interface

Our example considers a common problem concerning an authentication step present in various everyday interactive systems, e.g. internet banking. Before any transaction, users must establish their identities by providing a user name and a login password. The system checks whether the provided password is the same as the one associated with the provided user name and stored in the system's database. On the surface, one could expect the design of an authentication interface to be simple, e.g. like the one in Fig. 3(left). In reality, the situation is more complicated. The sizes of interface windows in internet banking systems are not fixed; users might change them at any time. This means that the layout of input fields is determined by an algorithm. Depending on this algorithm, the layout shown in Fig. 3(right) is possible when the window size is reduced. We will argue that the two interfaces are not equally secure and will show how confidentiality leaks in the second one can be detected using our verification framework.

We assume that a high security channel is associated with login passwords and a low security channel with user names. This could mean, e.g., that the text entered into the name box is echoed on the screen whereas an entry into the password box is not. The data is sent whenever the users press the `Enter` button. The operation of the authentication mechanism is illustrated by a finite state machine in Fig. 4. We distinguish two cases of incorrect input represented by the transitions `IncompleteData` and `IncorrectData`. The authentication procedure moves into the `INCOMPLETE` state when `Enter` is pressed and either a user name or a password is missing from the input boxes. An appropriate error message is displayed by the interface, and no other options for the user are given until the message is acknowledged. Once the user acknowledges it, the authentication procedure returns to the `INIT` state. The transition `IncorrectData` represents the case when both a user name and a password are provided but some of this data is incorrect. Upon acknowledgment, the procedure moves into the `WARNING` state in this case. The idea is that, for security reasons, a single authentication attempt with incorrect data is allowed before the authentication procedure aborts the interaction (`STOP` state). Finally, authentication succeeds when the user provides correct data, represented by the `LOGIN` state reachable from either the `INIT` or `WARNING` state.

The SAL specification of the authentication procedure is a direct translation of the diagram in Fig. 4. The two input boxes are modelled as the type `Inbox = {A, B}`. Each box has a number of attributes: position, security level, "visibility", label, text entered and text displayed, modelled as arrays with the range `Inbox`. Thus, `position[j]` is a record with the coordinate fields `x` and `y`, denoting the top-left corner of box `j`. Its width and height are represented by the constants `dx` and `dy`. The security level `level[j]` is either `Low` or `High`; `displayed[j]` indicates whether `j` is displayed (visible) or not. The label `label[j]` is a value of type `{NameLabel, PasswordLabel}`. Finally, `value[j]` and `display[j]` represent the text entered and displayed, respectively. The array `value` and booleans `EnterPressed` and `Acknowledged` are the inputs of the authentication procedure, whereas `position`, `displayed`, `label`, `display`, and booleans `DisplayedEnter`, `IncompleteMsg`, `IncorrectMsg`, `LastAttemptMsg`, `LoginMsg`, and `AbortMsg` are its outputs.

4.2 A User Model

Now we instantiate the generic module `User` for the authentication task. We start by specifying the state space of the concrete user model. For each input box `j`, we assume that a person either sees it or not, and perceives its label and the text displayed, represented by `seen[j]`, `label[j]` and `value[j]`, respectively. The perception of whether the `Enter` button is active is denoted by `EnterActive`. The person also perceives whether an error, warning or authentication message is given, denoted by `ErrorMsg`, `WarningMsg` and `LoginMsg`. Variables `InputName` and `InputPass` denote the perception of which of the two boxes prompts for the user name and which for the password. Finally, `name` and `password` denote the values the person perceives as a user name and password. All these components form a record type, `In`, which is used to instantiate the corresponding type variable in `User`.

Next, we specify variables related to the actions users might take. The text typed into box `j` is represented by `value[j]`. The booleans `EnterPressed` and `Acknowledged` denote whether the `Enter` button and a button to acknowledge messages are pressed. These components form a record type, `Out`. We assume users remember their user name, `name`, and login password, `password`. They also remember whether they already typed information into box `j`, denoted `entered[j]` (reset to false when an error message is acknowledged), and keep track of whether there was a failure to authenticate, denoted `failed`. These form a record type, `Mem`, which also records the actions taken in the previous step in a component of the type `Out`. The reality surrounding our system is given by a record type, `Env`. It includes the user name, `name`, and the correct password, `password`.

We assume that user knowledge of authentication includes the need to communicate (1) user name and (2) login password. This knowledge is specified as user goals (elements of array `goals`) instantiated by giving the action guard and the update to the output component. For the goal of communicating the user name, the guard is that an input box, regarded as the name box, is seen. The output action is to type in the user name as the person perceives it:

```
grd := λ(in,mem,env):  in.seen[in.InputName]
tout := λ(in,out0,mem):λ(out):  out = Default(out0.value)
                                WITH .value[in.InputName] := in.name
```

where `Default(x)` is a record with the field `value` set to `x` and all other fields set to false thus asserting that nothing else is done. The memory update (omitted) simply records the action taken. As an example, we will specify the most complicated memory update below. The action of communicating the login password is specified similarly. Since the environment updates change nothing (in all the actions), they are omitted here.

We assume that the user can *react* to the active `Enter` button by pressing it. For this to happen, the user must not have the recollection of a failure to authenticate. Alternatively, if there was such a failure, we expect the user to be more careful and press `Enter` only when both input boxes were filled in:

```
grd := λ(in,mem,env):  in.EnterActive ∧ (NOT(mem.failed) ∨
          (mem.entered[in.InputName] ∧ mem.entered[in.InputPass]))
tout := λ(in,out0,mem):λ(out):
           out = Default(out0.value) WITH .EnterPressed := TRUE
```

We also assume that the user can acknowledge error messages. This only happens when the message is interpreted as an error signal. The acknowledgment must also not have occurred, as indicated by the memory, in the previous step. By acknowledging the error message, the user records in the memory the fact of a failed authentication attempt, and “forgets” previously typing data into the input boxes (since the data was rejected), formally specified as:

```
grd := λ(in,mem,env):  in.ErrorMsg ∧ NOT(mem.out.Acknowledged)
tout := λ(in,out0,mem):λ(out):
           out = Default(out0.value) WITH .Acknowledged := TRUE
tmem := λ(in,mem0,out):λ(mem):  mem = mem0 WITH .failed := TRUE
           WITH .entered := [[j:Inbox] FALSE] WITH .out := out
```

As discussed earlier, the need to communicate the name and password is modelled as user goals. However, it is plausible that the user makes an error when trying to achieve those goals, e.g. enters a wrong password or presses `Enter` when some box is empty. Errors can also occur due to user habits; relying on previous experience, the user might expect the input box for the name to precede that for the password. In such cases, once the error message has been acknowledged, the system prompts for a new authentication attempt. We assume that the user will respond to this prompt. The response is modelled as two reactive actions. In the case of the password, the action guard is that an input box is seen (as for the corresponding user goal) and the password was not entered, as indicated by the memory, in the previous step. The output action is the same as for the corresponding user goal. Finally, the memory update records the fact of entering the password:

```
grd := λ(in,mem,env):  in.seen[in.InputPass] ∧
                       NOT(mem.entered[in.InputPass]) ∧ mem.failed
tout := λ(in,out0,mem):λ(out):  out = Default(out0.value)
                         WITH .value[in.InputPass] := in.password
tmem := λ(in,mem0,out):λ(mem):  mem = mem0
           WITH .entered[in.InputPass] := TRUE WITH .out := out
```

The reactive action for entering the name is analogous to the one above.

Goal and wait predicates are the last parameters used to instantiate the `User` module. The display of `LoginMsg` confirms authentication which is the main goal. We also assume that there are no signals that a user could perceive as a suggestion to wait. These predicates are specified as follows:

```
PerceivedGoal = λ(in,mem): in.LoginMsg
Wait = λ(in,mem): FALSE
```

Finally, the user model for the authentication task, `UserAuthenticate`, is defined by instantiating the generic user model with the parameters (goals, reactive actions, perceived goal and wait condition) just defined.

4.3 User Interpretation

So far we have specified an authentication interface and have developed a formal model of its user. As in reality, the state spaces of the two specifications are distinct. The changing interface state is first attended to then interpreted by the user. Next we specify this interpretation, thus connecting distinct state spaces. The specification is given as a new SAL module, `interpretation`. The module, being a connector, has input variables that are the output variables of the interface, and an output variable that is the input (perception) component (record `in`) of the `UserAuthenticate` module.

In the authentication task, the crucial aspect of user interpretation is the perception of the meaning (function) of the two input boxes. Their function is indicated by labels, however, we assume that people may not pay sufficient attention to the labels. Instead, the user might assume the name box comes first. The perception of precedence depends on the layout (coordinates) of boxes in the interface window. Formally, we define the condition when the input box `i` precedes `j` as follows (`pos` is an array of coordinates):

```
precedes(i,j,pos) = (pos[i].x + dx < pos[j].x ∧ pos[i].y ≤ pos[j].y)
                  ∨ (pos[i].x ≤ pos[j].x ∧ pos[i].y + dy < pos[j].y)
```

Intuitively, this means that `j` is placed to the right and to the bottom of `i`. Thus, the name box in the left interface in Fig. 3 precedes the password box, whereas neither of the boxes in the interface on the right precedes the other.

Now we formally define the user interpretation of the function of input boxes, depending on their layout and labelling. We distinguish three cases. First, the user might judge the function of input boxes from their labels:

```
ByLabel(l,x) = ∃(i,j):  l[i] = NameLabel ∧ l[j] = PasswordLabel ∧
                        x.InputName = i ∧ x.InputPass = j
```

Second, if `i` precedes `j` then `i` is perceived as a name and `j` as password box:

```
ByPrecedence(pos,x) = ∃(i,j):  precedes(i,j,pos) ∧
                               x.InputName = i ∧ x.InputPass = j
```

Finally, the user might get confused. This is possible when neither of the input boxes precedes the other or their labels are the same; the judgment about the function of the boxes is random in this case:

```
Random(pos,l,x) = (l[A] = l[B] ∨ ∀(i,j):NOT(precedes(i,j,pos))) ∧
                  x.InputName ≠ x.InputPass
```

User interpretation is modelled as a SAL definition which allows one to describe system invariants. Intuitively, this means that the left-hand side of an equation is updated whenever the value of the right-hand side changes. We assume that, once the user makes a mental commitment to a goal or reactive action, the interpretation of the interface outputs does not change until the associated physical action is performed. If there is no commitment, the user directly perceives the `Enter` button, the displayed input boxes with their labels and displayed text, and the interface messages. Hence the first seven conjuncts simply rename the interface variables to the corresponding fields of the record `in` in the definition below:

```
DEFINITION in ∈ { x:In | IF NOT(gcomm ∨ rcomm) THEN
    x.EnterActive = DisplayedEnter ∧
    x.seen = displayed ∧ x.label = label ∧ x.value = display ∧
    x.ErrorMsg = (IncompleteMsg ∨ IncorrectMsg) ∧
    x.WarningMsg = LastAttemptMsg ∧ x.LoginMsg = LoginMsg ∧
    x.name = IF x.WarningMsg THEN env.name ELSIF mem.name ENDIF
    x.password = IF x.WarningMsg THEN env.password
                 ELSIF mem.password ENDIF
    IF x.WarningMsg THEN ByLabel(label,x)
    ELSIF MajorChanges(p,position,l,label) THEN ByLabel(label,x)
       ∨ ByPrecedence(position,x) ∨ Random(position,label,x)
    ELSE x.InputName = s.InputName ∧ x.InputPass = s.InputPass ENDIF
  ELSE x = s ENDIF }
TRANSITION s′ = in; p′ = position; l′ = label
```

For the user name and password, the user relies on the memory unless a warning message is displayed. If so, we expect the user to be careful enough to provide the correct values. For simplicity, here we do not consider how this is actually achieved (perhaps they are taken from a notebook), assuming that the values from the environment specification are used.

As explained earlier, the perception of which of the two boxes is for the names and which for the passwords is more complicated; the results of this perception are assigned to `InputName` and `InputLabel`, respectively. We assume that, upon receiving a warning message, the user becomes more careful and interprets the input boxes by their labels. Otherwise, if there are major changes in the layout of the boxes, the interpretation is an arbitrary choice between the three cases defined above. If there are no major changes, the interpretation of the boxes is the same as in the previous step. The auxiliary variables `s`, `p` and `l` are not intended to represent aspects of cognition. Intuitively, they, and the related `TRANSITION` section, are used to store the previous interpretation which allows specifying that user interpretation does not change. Finally, "major" changes mean changes in the precedence of input boxes or any label change:

```
MajorChanges(pos0,pos,l0,l) =
     ∃(i,j):  precedes(i,j,pos0) ≠ precedes(i,j,pos) ∨ l0[i] ≠ l[i]
```

Admittedly, this attempt to formally specify how the user perceives input boxes already hints at potential problems, even before the actual verification. However, we aim at developing a generic model of interpretation which would turn the specification process into a simple instantiation of the generic model.

4.4 Verification

We have specified an authentication interface and its user model. Now the correctness properties of this interactive system can be analysed. We start from the interface with no constraints on the layout of the input boxes (other than that they do not intersect). The usability property (1), the user eventually achieving the perceived goal, is satisfied by the interactive system. Next we proceed with the analysis of security aspects of the system.

Even though security properties for each concrete system can be specified separately, we prefer to take a generic approach as with the user model itself. We thus introduce a generic module, `tester`. The idea is that the module, composed with an interactive system, monitors the communication between the device and the user. When security is breached, it sets the variable `SecurityBreach` to true. What security aspects are monitored is determined by the instantiation of the module. It has three parameters. The type variable `Chan` represents the communication channels. The predicate `filter` specifies which of the channels are monitored. Finally, `test` denotes security sensitive data. When this data appears on a monitored channel, `SecurityBreach` is set to true. The transitions of the module are the following family of commands:

```
[](j:Chan):  filter(j) ∧ value[j] = test  →  SecurityBreach′ = TRUE
```

For our authentication task, `Chan` is instantiated to the two input boxes. The security sensitive data is the actual user password `env.password`. Finally, the monitored channels are low security ones: `filter(j) = (level[j] = Low)`. With this instantiation of `tester`, we check property (3); the verification fails. The counterexample produced by SAL indicates that the user enters the password into the name box. The analysis of the specifications reveals that this counterexample occurs because neither of the boxes precedes the other which confuses the user.

Why was this confusion not detected when verifying property (1)? The answer is that it does not prevent the user from achieving the main goal, authenticating their identity. Our user model is "smart" enough to recover from the mistake made due to this confusion and, after receiving a warning message, to provide the required information according to the labels of the input boxes. Such a recovery, however, is not good enough from the security point of view, since the mistake could have already breached security and undoing or redoing a wrong action cannot undo the consequences of this breach in most cases, which is detected by the failure to establish property (3).

How can we avoid this security breach? Since the confusion leading to it is caused by the layout of the input boxes, the solution is to display them as in Fig. 3(left). However, one must be careful even with such a layout. If the password box preceded the name box, people might enter their password into the name box, due to their habits rather than confusion. Again, this security breach is detected by verifying (3). This property holds when the layout of the input boxes is such that `precedes(InputName,InputPass,position)` is true.

5 Conclusion

In previous work, we assumed that usability verification is enough to establish user-centred correctness of interactive systems. This is not true within security- or mission-critical contexts where it is possible to achieve the main goal while exposing ourselves to various security or safety risks. Here we addressed one aspect of interactive systems security – information-flow security. We discussed how security breaches can be detected using our earlier framework. The main focus, however, was an extension of that framework to address security aspects more directly and link them to the usability properties. For this, we introduced into our framework a generic tester module which monitors information flow between the user and the device and registers security breaches in it. Using the tester, we added to our verification approach a correctness property which captures some security aspects of interactive systems.

To illustrate these extensions, we considered a simple authentication interface. We showed how the layout of input fields combined with user habits can influence the user (mis)interpretation of interface prompts, possibly leading to confidentiality leaks (see our SAL specs at `http://www.dcs.qmul.ac.uk/~rimvydas/usermodel/security.sal`). We demonstrated how these leaks are detected using the SAL verification tools, and how the analysis of the SAL counterexamples can help in eradicating cognitively susceptible interface designs. The SAL environment was primary chosen because of its support for higher-order specifications. This is necessary for developing a generic cognitive architecture as ours.

The user interpretation stage was introduced into our framework from general considerations. We previously showed how modelling user interpretation allows us to detect usability problems due to the shape-induced confusion over device prompts [14]. Here we showed that similar ideas apply in the context of security properties and their dependance on the layout of input fields. Finally, we also considered user habits, which we had not dealt with before.

Human factors in the security context have been considered before [1,11]. The novelty of our approach is dealing with the cognitive aspects of security in a completely formal way, making them amenable to automatic verification. Moreover, our cognitive architecture could be used to prove generic results on, e.g., design rules for security, using its formalisation within the HOL prover.

References

[1] Adams, A., and M. A. Sasse, *Users are not the enemy*, CACM **42**(12) (1999), 41–46.

[2] Beckert, B., and G. Beuster, *A method for formalizing, analyzing, and verifying secure user interfaces*, in press: Proc. ICFEM 2006, LNCS, Springer, 2006.

[3] Blandford, A., and G. Rugg, *A case study on integrating contextual information with usability evaluation*, Int. J. Human-Computer Studies **57**(1) (2002), 75–99.

[4] Butterworth, R., A. Blandford, and D. Duke, *Demonstrating the cognitive plausibility of interactive systems*, Form. Asp. Computing **12** (2000), 237–259.

[5] Byrne, M. D., and S. Bovair, *A working memory model of a common procedural error*, Cognitive Science **21**(1) (1997), 31–61.

[6] Cerone, A., P. A. Lindsay, and S. Connelly, *Formal analysis of human-computer interaction using model-checking*, in: Proc. SEFM 2005, IEEE Press, 352–362.

[7] Curzon, P., and A. E. Blandford, *Detecting multiple classes of user errors*, in: R. Little, and L. Nigay, eds., Proc. EHCI 2001, vol. 2254 of LNCS, Springer, 2001, 57–71.

[8] Goguen, J. A, and J. Meseguer, *Security policies and security models*, in: Proc. IEEE Symp. on Security and Privacy, Apr. 1982, IEEE Press, 1982, 11–20.

[9] Hollnagel, E., "Cognitive Reliability and Error Analysis Method," Elsevier, 1998.

[10] John, B. E., and D. E. Kieras, *The GOMS family of user interface analysis techniques: Comparison and contrast*, ACM Trans. CHI **3**(4) (1996), 320–351.

[11] Ka-Ping, Y., *User interaction design for secure systems*, in: R. Deng, et al., eds., Proc. ICICS 2002, vol. 2513 of LNCS, Springer-Verlag, 2002, 278–290.

[12] de Moura, L., S. Owre, H. Ruess, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, *SAL 2*, in: R. Alur, and D.A. Peled, eds., Computer Aided Verification: CAV 2004, vol. 3114 of LNCS, Springer-Verlag, 2004, 496–500.

[13] Newell, A., "Unified Theories of Cognition," Harvard University Press, 1990.

[14] Rukšėnas, R., P. Curzon, J. Back, and A. Blandford, *Formal Modelling of Cognitive Interpretation*, in press: Proc. DSVIS 2006, LNCS, Springer, 2006.

[15] Rushby, J., *Analyzing cockpit interfaces using formal methods*, Electronic Notes in Theoretical Computer Science **43** (2001).

[16] Sabelfeld, A., and A. C. Myers, *Language-based information-flow security*, IEEE Journal on Selected Areas in Communications **21**(1) (2003), 1–15.

[17] Zurko, M. E., *User-centered security: Stepping up to the grand challenge*, in: Proc. ACSAC 2005, IEEE Press, 2005, 187–202.