Formal Methods for Industrial Products

Susan Stepney[1] and David Cooper[1] *

Logica UK Ltd, Betjeman House, 104 Hills Road, Cambridge, CB2 1LQ, UK
stepneys@logica.com cooperd@logica.com

Abstract. We have recently completed the specification and security proof of a large, industrial scale application. The application is security critical, and the modelling and proof were done to increase the client's assurance that the implemented system had no design flaws with security implications. Here we describe the application, specification structure, and proof approach.

One of the security properties of our system is of the kind not preserved in general by refinement. We had to perform a proof that this property, expressed over traces, holds in our state-and-operations style model.

1 Introduction

Over the past few years we have been working with the National Westminster Development Team (now *platform seven*), proving the correctness of Smartcard applications for electronic commerce, which are currently being sold as commercial products.

We have modelled the abstract security behaviour and properties of the products, modelled the more concrete top level design, and have rigorously proved the preservation of both functional and non-functional security properties. All work was done in Z.

We have previously described one of the Smartcard products, an electronic purse [Stepney *et al.* 1998]. Here we describe another product: a smartcard operating system that ensures a secure environment for running segregated applications.

2 Overview of the application

A Smartcard operating system should host, and segregate, separately loaded executable *applications*. If no loaded application can interfere with any other applications co-resident on the smartcard, independent application providers can be assured that their own applications are operating in a secure environment.

NatWest called in Logica to discover if it is feasible in a commercial setting both to develop formal models of such a system and its security policy, and to prove that the system design meets all the security properties required.

* current address: Praxis Critical Systems Ltd, 20 Manvers Street, Bath, BA1 1PX
cooperd@praxis-cs.co.uk

3 Special features

There were various things we had to deal with, to ensure that we were specifying and proving the right things, and to ensure that we were presenting the specification and proof to the right level of abstraction, detail and clarity. Some of these issues are discussed in the following sections.

3.1 Security models and proofs

We specified two key formal models, an abstract *Security Policy model* (SP) that clearly captures the desired security properties of the system, and a more concrete *Hardware Design model* (HW), that clearly maps to the semiformal design. We also performed a proof that HW exhibits the security properies of SP.

We actually chose to structure the model in three levels: *SP*, *VM* (virtual machine, an intermediate level), and *HW*, introducing implementation detail only where needed, and as low down the specification hierarchy as possible. Having the intermediate model resulted in more proofs, but simpler ones.

3.2 Segregation with communication

The fundamental security property of Smartcard operating systems is that applications are *segregated*; one application cannot read or change secret data in another application, either by rogue intent or because of a bug. However, segregation need not be absolute; there could be support for applications to communicate with each other *to some limited extent*, over explicitly identified overt communication channels.

Although much has been published in this area (see, for example, [Bell & Padula 1976], [Rushby 1981], [Goguen & Meseguer 1984], [Bell 1988], [Jacob 1992], [Roscoe 1995], [Gollman 1998], among many others), nothing existing fitted our needs without modification, because of other technical constraints imposed on the particular commercial product we were dealing with. (Industrial scale formal methods work often requires modification of, or extension to, existing idealised academic results, because of conflicting real world constraints and demands.) So, building on the existing concepts, we formulated a suitable property of *segregation with communication* [Cooper & Stepney 2000] and proved that our system model possesses an appropriate instantiation of this property. Developing a suitable formulation, and proving it holds, was the major technical challenge of our development work.

Summarising that definition: The segregation property is formulated as constraints on sets of system traces (sequences of communication events) that ensures that the system's applications are behaving independently, except for the explicitly identified communication events. The segregation property states that if certain event traces are allowed, then other event traces, corresponding to the same applications executing in a different order, must also be allowed, because

the only way the other traces could be *disallowed* would be by some covert communication, coordination, or interference between the applications.

This segregation property is a kind of property not preserved in general by refinement. Refinement can be viewed as taking a subset of allowed system traces, provided the subset does not narrow the precondition, but just resolves non-determinism. Yet a property that says 'if t_1 is a trace of the system, then so is t_2' is not necessarily preserved by a subset of the traces (this would correspond to using some covert communication to resolve the non-determinism). So, as well as proving that HW is a refinement of SP (necessary to show that HW has the functional properties of SP), we have to provide a different kind of proof to show that HW also preserves the segregation property of SP.

4 Modelling consequences

Because of these special features of the problem, we did not have complete freedom in the way we could specify the model. We had to structure it to allow the various proofs to be performed.

4.1 Segregation and multi-promotion

The requirement for segregation with communication permeates the entire structure of our specification and proof approach.

Although the segregation property is important, Smartcard operating systems have a lot of other, functional, properties and behaviour that must be specified. These are most naturally captured in Z using a conventional state-and-operations specification style, and SP–HW model correspondence shown with a conventional data refinement proof. In addition, any mapping from the concrete HW model to a semi-formal design is more naturally achievable for a state-and-operations style specification.

These considerations led us to adopt such a style. But then we needed a way of expressing the trace-based segregation property as a property of our state-and-operations model. We proved an *unwinding theorem* [Cooper & Stepney 2000], which allowed us to show that (a particular form of) *unconstrained multi-promotion* has our segregation property.

Promotion is a commonly used Z specification structuring technique that allows operations specified on individual 'local' pieces of state to be 'promoted' to operations on a 'global' state comprising labelled copies of the local state (explained in [Barden *et al.* 1994, chapter 19]). A 'framing schema' identifies the single piece of local state being changed, and requires the other pieces of local state to be unchanged. The framing schema is then combined with the relevant local operation schema (and the local state hidden) to say *how* that identified local state changes.

Multi-promotion is an obvious extension to allow an operation to affect two or more pieces of local state in concert. In a Smartcard operating system with communicating applications, there is the need to promote a single application,

or two applications, or three applications, or all loaded applications, on occasion. For the case of two pieces of local state promoted together, with a global state like $global : ID \pfun Local$, the relevant framing schema might look like

$$
\begin{array}{|l}
\hline
\Phi Two \\
\Delta Global \\
\Delta Local \\
\Delta Local_2 \\
from?, to? : ID \\
\hline
\mathsf{disjoint}\ \langle \{from?\}, \{to?\} \rangle \\
\{from? \mapsto \theta Local, to? \mapsto \theta Local_2\} \subseteq global \\
global' = global \oplus \{from? \mapsto \theta Local', to? \mapsto \theta Local_2'\} \\
\hline
\end{array}
$$

and then the corresponding promoted operation would look like

$$GlobalOp \mathrel{\widehat=} \exists\, Local; Local_2 \bullet \Phi Two \land LocalOpFrom \land LocalOpTo_2$$

In general, multi-promotion allows an arbitrary choice of the number of application promoted.

Unconstrained multi-promotion is a promotion where there are no global constraints on the promoted state or operations, that is, no constraints linking pieces of local state.

Informally: because all the operations are defined in terms of *local* (single application) state only, there are no opportunities for one local state's behaviour to be influenced by another's at the point of specification, so no communication can occur. Formally: our unwinding theorem proves that this is the case.

Security properties can have notoriously counter-intuitive consequences, so we were very careful to prove our property formally, rather than relying on informal justifications. It is relatively easy to justify that an operation on one piece of state does not *alter* another piece of state: that other piece of state can be seen to be unchanged. It is much harder to justify that every operation changes its piece of state in a way that is independent of the value of all other segregated pieces of state: the other states do not change, but their values are accessible through the mathematical formulation. Our formalism not only made precise what was meant by segregation with communication, but also formally justified that unconstrained multi-promotion, with its explicit communication between the promoted states, exhibits this form of segregation.

Unfortunately, the formulation of our unwinding theorem, although a kind of multi-promotion, is not expressed in the form most natural for a Z specification. It is centered around the communication events, and those particular applications involved in the event have to be deduced. On the other hand, the conventional Z multi-promotion style illustrated above, identifies explicitly which applications are involved in a particular promoted operation, and the corresponding communication event must be deduced.

We had two choices in order to prove our system model to be segregated: either write our specification directly in the communication-centred form suited to the segregation theorem, or prove that our specification written in the more natural application-centred form was equivalent to one formulated in terms of communications.

After experimenting with both approaches, we decided on the second one. Although that choice requires performing an extra proof, we felt that the added clarity of the specification, and ease of proof in other areas, outweighed the penalty.

4.2 Modelling the functionality

When talking about an operating system supporting applications, one most naturally thinks of the OS as a 'layer' *beneath* the applications. However, that natural modelling approach is not compatible with our view of segregation, which is expressed in terms of applications only.

Smartcard operating systems allow user applications to be securely loaded, securely deleted, and securely executed. In addition to loading and deleting user applications, and mediating between inter-user-application communication, Smartcard operating systems usually offer some additional trusted functionality, such as random number generation and various query functions. We needed to incorporate such functionality into our segregation framework, which recognises only 'applications'. In addition, ISO standard Smartcards have some required functionality (Master File, ATR File and Directory File) that behave to some ways like user applications – they are selectable – but not in others – they have fixed functionality and are permanently resident.

So we modelled loadable and deletable *user applications*, we modelled ISO standard functionality as three special applications, and we modelled the operating system functionality itself as the single *Scos trusted application*.

In order to simplify the segregation proof, which talks of a single kind of application, we used a free type to build applications from user applications, ISO applications, and the *Scos* application. Also, because user applications are loadable and deletable, but the segregation formulation assumes the segregated applications are fixed (it assumes a total mapping $APPL \longrightarrow LocalState$), we also modelled absent user applications.

$$APPL ::= scos\langle\langle Scos\rangle\rangle \mid iso\langle\langle ID\rangle\rangle \mid user\langle\langle UserAppl\rangle\rangle \mid absent\langle\langle ID\rangle\rangle$$

This formulation using a total mapping over a free type does make the specification a little clumsy in places (particularly the continual extraction of states from their free type wrappers), but makes it possible for us to prove the segregation property.

5 Determinism

We used determinism in two different places to solve two different modelling problems.

5.1 Imposed determinism

We had to make sure our *SP* model is sufficiently constrained not to allow unwanted refinements; in particular, that any resolution of non-determinism does not subvert certain confidentiality requirements (this is in addition to the requirement that the refinement also preserves the segregation property). The *Scos* is intended to be a *trusted* application: trusted by the other applications not to pass any of their communications with it to other applications. For example, the *Scos* application can be trusted not to store the last random number it gave to application A, then use that as a way of resolving a non-deterministic interaction with application B, and it can be trusted not to store the messages that it passes between an application and the external communication channels.

So, in our *SP*, the *Scos* application's behaviour must be tied down sufficiently that it can be seen to be trustworthy. We achieved this by making the abstract *Scos* state small, and *imposing operational determinism*. Thus it is not possible to use secret or covert information to resolve the nondeterminism in a more concrete model. Because the specific behaviour of user applications is not specified in the abstract *SP* model, it is not possible to make that model explicitly deterministic. Instead, we added a predicate to assert that every operation's behaviour *is* deterministic, without specifying *what* that behaviour is; we introduced a requirement for functionality into the system behaviour.

If the abstract state is captured by the schema A, and the operation by the schema AOp, we can define a function that converts operations defined using delta schemas into operations defined as relations between (before and after) schemas, as:

$$\begin{array}{|l} relA : \mathbb{P}\, AOp \longrightarrow (A \times IN) \leftrightarrow (A \times OUT) \\ \hline \forall\, op : \mathbb{P}\, AOp \bullet relA\ op = \{\, AOp \mid \theta AOp \in op \bullet (\theta A, m?) \mapsto (\theta A', m!) \,\} \end{array}$$

Assume the non-determinised form of the (total) operation is AOp. We can define possible deterministic forms as[1]

$$aDet == \mathbb{P}(relA\ AOp) \cap (\_ \longrightarrow \_)$$

and an augmented state as

$$ADet \mathrel{\widehat=} [\, A; f : aDet \,]$$

The deterministic operation is then

[1] In general, this is a sufficient, but not necessary, constraint for determinism, as discussed in section 5.3. In this case, however, our *abstract* model is "sufficiently abstract" in that all its state is observered, either through outputs or finalisation. (Technicalities aside, a merely sufficient condition for determinism is sufficient for our pragmatic, industrial purposes.)

$AOpDet$

$\Delta ADet$
$m? : IN$
$m! : OUT$

$f' = f$
$f(\theta A, m?) = (\theta A', m!)$

This says that the particular choice of determinism, f, is unchanged by the operation, and that the operation behaves like AOp and is deterministic in the way captured by $aDet$.

Only the operations of the SP model are constrained to be deterministic in this way. The initialisation is highly non-deterministic, because it does not constrain the value of f. Refinement chooses the particular deterministic behaviour that is implemented. We proved our refinement using the conventional 'forward' Z refinement rules ([Spivey 1992b, chapter 5], augmented with finalisation [Stepney *et al.* 1998]), and not the 'backward' Z refinement rules, thus showing that the refinement did not move the non-determinism at initialisation to occur later [2]; the refined operations stay deterministic, and so the refined *Scos* application remains trustworthy.

If it were impossible to make the operations deterministic (if $aDet$ were empty), adding such a constraint to $ADet$ would make the state empty. We proved this was not the case by showing the existence of such an f expressed in terms of a more concrete model proved both to be deterministic (including initialisation) and to be a refinement of the non-determinised abstract model. This concrete model is then a suitable refinement of the (non-empty) determinised abstract model.

5.2 Determinism and refinement

We used our unwinding theorem and multi-promotion to prove that our top-level SP and intermediate-level VM models are segregated. However, our lowest level HW model is not structured as an unconstrained multi-promotion: it is highly

[2] An uninterpreted Z specification is not sufficient by itself to define the legal implementations. For example, it may not be clear what schemas are intended to correspond to operations to be implemented, and which are merely scaffolding. The most common difference in interpretation is behaviour outside the precondition: [Spivey 1992b, chapter 5]'s forward rules allow "weakening the precondition", whereas different Z refinement rules need to be used to support a firing condition interpretation [Josephs 1991].

It is customary in Z specifications to leave much of this interpretation implicit; we were more careful, stating (necessarily informally) precisely what schemas comprised the operations, which refinement rules we were using, and why. All this kind of validation and meta-argumentation, that the right property is being proved, and that the proof performed really does establish that property, is carefully documented for the reviewers' scrutiny.

constrained, because it shows how the applications are arranged in a flat memory space, and how they share use of the RAM. Such a constraint is the very kind of thing that might indicate a covert communication through the shared memory. We had to prove that the applications as laid out in a single flat memory space remain segregated. We needed another way to demonstrate segregation.

Segregation is expressed in terms of traces; refinement in terms of subsets of traces. If a model is deterministic, a refinement cannot resolve non-determinism (remove traces). If a model is also total, a refinement cannot weaken the pre-condition (add traces). So a refinement of a total, deterministic model has the same traces: if the original is segregated, so must be the refinement. We used this fact to prove segregation of our *HW* model: by proving segregation (by unconstrained multi-promotion), totality, and determinism of our *VM* model.

5.3 Determinism and traces

Whilst trying to solve the problem of making the *Scos* provably trustworthy, and proving the *VM* and *HW* have the same traces, we found ourselves bandying about phrases like 'operationally deterministic', yet when we came to write down the proof obligations, we realised our first naive attempt was too restrictive. We had thought we had to prove that the state transition is functional, to prove that

$$BOp \vdash relB\ BOp \in B \times IN \pfun B \times OUT$$

(where *relB* is defined with respect to state B and operation BOp in a similar way to *relA* above[3]).

However, consider a specification of a state comprising a set, with some obviously deterministic operations on it such as 'add an element' and 'remove an element'. It would be quite legitimate to refine this set to a sequence, and the operation of 'add an element to the set' to 'if it is not already there, add the element *anywhere* in the sequence'. The abstract state transition relation is functional, but the concrete state transition relation is no longer functional, yet the observed behaviour is still deterministic.

And this corresponds closely to the case of a Smartcard operating system: in the concrete *HW* model there are various possible ways of laying out applications in memory, but the observed behaviour is independent of which way this layout is actually implemented. Our naive proof obligation was too strong. We were able to use our trace model of segregation to help us determine the appropriate proof obligation for determinism.

[3] It would be nice if such a relation could be defined generically. This is not possible in Z as it stands today, because generic definitions cannot be constrained to be applicable to particular sets, such as schemas. Type constrained generics [Valentine *et al.* 2000] would allow such a definition.

6 Functional and non-functional properties

6.1 Two security models

We needed to specify various *functional* security properties, based in the usual way on an external view (inputs and outputs) of the system. We also needed to express the segregation property, based on a necessarily internal view of the inter-application communications, not engaged in with the outside world. These two views need to be related somehow. We also had to make the specification structure match the implementation; in particular, cope with the fact that the overt inter-application communication channels are unobservable from outside the smartcard.

So we wrote *two* security policy models, one capturing the functional properties, SP_f, with only the external communications visible, and one capturing the segregation property, SP_s, with all the external and internal communications present. In principle, these models could be entirely unrelated. In practice, we made them very similar: they differ *only* in the external observability of the inter-application communication channels, which are fully visible in the SP_s model, and finalised away [Stepney *et al.* 1998] to invisibility in the SP_f model.

We also wrote two corresponding intermediate models, VM_f and VM_s, and proved that each captured the required properties of the corresponding SP model. We wrote a single concrete model, HW, corresponding to the implemented device, and proved it possessed the properties of both the SP models, via the VM models.

6.2 Differently segregated

Segregation is a property of a single system. Two different systems may each be segregated, yet have no relationship to each other. For example, once we have carefully specified our segregated SP, with lots of separate SP-applications not interfering, we do not want to be presented with a purported implementation that bundles the whole behaviour into a single HW-application, despite such a single-application system necessarily being 'segregated' according to our definition.

So we defined the property of *segregation with respect to* a model (*segWrt*) to capture the fact that the two models are segregated *in the same way*. This boils down to having corresponding applications communicating in the same way. We proved that a sufficient condition for B *segWrt* A, where A is segregated, is for B itself to be segregated with the same interpretation of application structure as A (that is, using the same *asEvent* bijection introduced in [Cooper & Stepney 2000]), and for B also to be a refinement of A.

Even that is not sufficient for our purposes, because we have two security models, SP_s defining the segregation structure, and SP_f, defining the visible functional behaviour. Our final concrete model HW is a necessarily a refinement of SP_f, not of SP_s, and so cannot be segregated with respect to it.

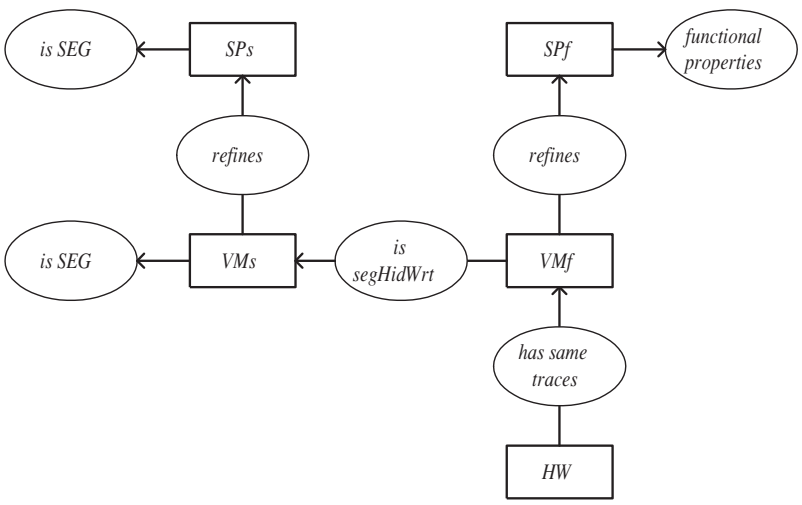

Fig. 1. Overview of relationships between the formal models (boxes) and the proofs (ellipses)

So we defined the property of *segregated and hidden with respect to* a model (*segHidWrt*), to capture the fact that one model *behaves as if* it is segregated in the same way as another, except that the communication channels are hidden. This corresponds quite nicely to the implementation, which is not physically segregated – applications share a flat memory space, and perform their allowed communications using shared memory buffers – yet nevertheless they behave *as if* they are segregated. Which is the whole point of the exercise, of course.

We also proved some properties about these relationships, in order to be able to prove that our concrete model has the desired segregation property. In particular:

$$\vdash C\ segHidWrt\ B_s \wedge B_s\ segWrt\ A_s \Rightarrow C\ segHidWrt\ A_s$$

7 Resulting specification structure

The specification and proof structure we developed is summarised in figure 1. This structure is intended to simplify the proofs. It also has the advantage that some of the details of the virtual machine can be changed without changing the model of security.

We prove that HW possesses the security properties, by proving both that it is a refinement of SP_f and that it is segregated and hidden with respect to SP_s

7.1 Abstract Security Policy model, SP

The abstract SP model describes the world of applications and their communication through explicitly identified overt communication *channels*. It expresses

some functional security properties to do with securely loading and deleting user applications, and the key non-functional property, that *applications are segregated*: that they do not communicate or otherwise interfere with each other, *except* over the overt channels.

Our *SP* model is relatively small, simple, and easy to understand, running to approximately 40 pages of Z and natural language commentary[4]. The difference between the SP_f and SP_s models is captured in two different finalisation schemas.

The simplicity of the *SP* model allows these communication channels to be clearly identified, so that the client can easily verify that these channels are acceptable. 40 pages of Z may sound a lot for an abstract model, but most of the complexity was in the identification of the several overt communication channels present in the design.

The required functional security properties are proved to be consequences of the various SP_f operations. The SP_s model is constrained to be segregated, which gives us the segregation property by definition.

The behaviour of the virtual machine, and hence the behaviour of user applications, is not specified at this level. *No matter what* a user application does, the system is secure (segregated).

SP is secure, by definition.

7.2 Virtual Machine model, *VM*

Our more concrete *VM* model captures the behaviour of the Virtual Machine that ensures that abstract applications remain segregated. In practice, segregation is achieved by performing run-time memory access checks; this is the critical aspect of the *VM* specification.

Our *VM* model is more complicated than the *SP*, reflecting the design of the Virtual Machine. *VM* adds more design detail to *SP* by specifying the detailed behaviour of the virtual machine; it captures the actual behaviour of a user application, given its code.

This model is approximately 140 pages long, of which about 80 pages is a detailed description of the virtual machine. Again, the difference between the VM_f and VM_s models is captured in two different finalisation schemas.

7.3 Concrete hardware model, *HW*

Our concrete *HW* model captures the memory map of the design, showing how the segregated applications are securely implemented in a common flat memory space of physical RAM, ROM, and EEPROM, with shared use of the RAM.

Our *HW* model is approximately 20 pages long. It captures the memory structure explicitly; the operations are defined indirectly, in terms of the *VM* operations and the retrieve relation.

[4] The various page lengths quoted here give an indication of the relative effort involved in each of the specification and proof sections. The actual effort involved was not inconsistent with the metric discovered in [Barden *et al.* 1992].

8 Resulting proof structure

8.1 Proof tree

Some of the arguments we have presented concerning segregation, determinism, and refinement are subtle. In the morass of detail inherent in a large scale specification and proof, it would be easy to miss out some steps. As well as convincing ourselves we had not missed anything, we also had to make the proof structure comprehensible to third party reviewers.

We devoted two chapters of the final document just to documenting the proof structure. The first of these chapters is an overview of the structure, describing what needs to be proved, how the proofs are broken down into large components, and which proofs rely on other proofs (illustrated in sections 8.3 and 8.4 below).

The second chapter is a detailed proof tree, summarising the entire proof structure, showing what proofs are done where in the document, and demonstrating that everything that needs to be proved has been proved.

8.2 Proof sizes

All but one of the security properties of our abstract model are *functional*, and so are preserved by refinement. The segregation property is non-functional, and is not preserved in general by refinement.

So we rigorously proved that our concrete HW model is a *refinement* of our abstract SP model (thus proving it exhibits the functional security properties), and that the HW concrete model also segregates applications (thus proving it exhibits the non-functional security property).

The purpose of performing a proof is to greatly increase the assurance that the chosen design (the behaviour of the virtual machine, and the memory flattening) does, indeed, behave just like the abstract model. We chose to do rigorous proofs by hand, because our experience of existing proof tools is that current tools are not yet appropriate for a task of this size[5]. We did, however, typecheck the statements of the proof obligations and many of the proof steps using a combination of *f*UZZ [Spivey 1992a] (see appendix A) and Formaliser [Flynn *et al.* 1990] [Stepney]. All proofs were also independently checked by third party reviewers.

The proofs of the refinement obligations, the preservation of the segregation property, and the proofs of some model consistency obligations take approximately 280 pages. In addition, there are approximately 100 further pages of

[5] Each 'proof step' in our rigorous proof is fairly small for a hand proof, because of the requirement for checking by independent reviewers: we could not instruct them to do "several pages of (unspecified) algebra" for each step. So each step typically involves one (or a few) applications of a simple inference rule such as cut, one-point, Leibnitz, or of a Z toolkit law, or of a schema calculus law.

Our Z proof tool evaluation exercises show that each of these rigorous steps typically expands out to 20–100 elementary steps when performed with a tool such as CADiZ [Toyn 1996] (ignoring the steps needed to prove the toolkit law, where relevant).

formal derivation in support of the underlying theory of segregation with communication over overt channels.

We performed various consistency proofs and proofs that our SP model possesses the desired security properties. But the bulk of the proof work was showing that the HW model is consistent with the SP model: that it has the segregation property of the SP_s model, and the functional properties of the SP_f model.

8.3 HW has SP segregation property

We prove that HW *behaves as if* it is segregated in the same way as SP_s, except that the communication channels are hidden:

$$\vdash HW \; segHidWrt \; SP_s$$

We do this by introducing the intermediate VM_s and VM_f models, and using the property of *segHidWrt* that

$$C \; segHidWrt \; B_s \wedge B_s \; segWrt \; A_s \Rightarrow C \; segHidWrt \; A_s$$

which allows us to break the proof into two parts:

1. $\vdash HW \; segHidWrt \; VM_s$
   We show that the HW model is segregated in the same way as the VM_s model, except for the internal communications being hidden, by showing that the VM_f model is segregated in the same way as the VM_s model, and that the HW model has the same traces as the VM_f model.
   (a) $\vdash VM_f \; segHidWrt \; VM_s$
   This is easy to prove, because VM_f is equal to VM_s except for the internal communications being hidden by finalisation, which is the definition of *seqHidWrt*.
   (b) $traces \; HW = traces \; VM_f$
   The traces are the same if HW is a refinement of VM_f, and VM_f is total (so no traces can be added by widening a precondition) and deterministic (so no traces can be removed by resolving non-determinism).
      i. $\vdash VM_f \sqsubseteq HW$
      We define the HW model operations in terms of the VM model operations and a retrieve relation, and so the refinement holds by construction provided certain properties hold of the retrieve relation [Woodcock & Davies 1996, section 18.3]. That is, we prove that the local retrieve is functional from HW to VM_f, is total (covers the HW state), and is surjective (covers the VM_f state).
      ii. $\vdash isTotal \; VM_f$
      We prove the preconditions of all the VM_f operations are *true*.
      iii. $\vdash isDeterministic \; VM_f$
      We prove all the VM_f operations are functional.

2. $\vdash VM_s$ *segWrt* SP_s
   We show that the VM_s model is segregated in the same way as the SP_s model, by showing that it is segregated, and that it is a refinement of the SP_s model.
   (a) $\vdash VM_s$ *isSegregated*
   We prove that the VM_s model is equivalent to one written as an event-centric unconstrained multipromotion. The unwinding theorem from [Cooper & Stepney 2000] gives us that such a model is segregated.
   (b) $\vdash SP_s \sqsubseteq VM_s$
   We prove refinement.

8.4 *HW* has *SP* functional properties

We prove that HW is a refinement of SP_f:

$$\vdash SP_f \sqsubseteq HW$$

We introduce the intermediate VM_f model, and use transitivity of refinement to split the proof into two parts.

1. $\vdash SP_f \sqsubseteq VM_f$
   (a) We state and prove lemma 'squeeze', that shows that refinement is preserved under hiding the internal communications
   (b) We apply lemma 'squeeze' to $\vdash SP_s \sqsubseteq VM_s$, proved above, 8.3 2(b)
2. $\vdash VM_f \sqsubseteq HW$
   proved above, 8.3 1(b)i

9 Results

As well as providing a specification and proof that helped our customer gain assurance about the security of their product, the use of formality and proof improved the design and exposed some problems.

9.1 Design of the virtual machine

A major part of a Smartcard operating system's security functionality is provided by its virtual machine: this performs appropriate run-time memory access checks to ensure applications access only their own memory.

The formal specification work proved that the designed checks are indeed sufficient to ensure segregation. But in addition, the formalisation of the checks fed back into the documentation of the virtual machine, documenting more clearly, uniformly and precisely what checks are needed.

The formal modelling was a valuable part of the iterative design process. To start with, formality was used as a thinking aid, as we and the design team used a Rapid Application Development approach [DSDM Consortium] to sketch the design of the virtual memory model and opcode structure. The formal work then

became more detailed, and the particular memory access checks were specified in detail.

In addition, the specification work exposed a flaw in the original design of one of the opcodes: in certain rare circumstances it could overflow and overwrite memory outside its allowed region. The opcode was redesigned to remove the flaw.

9.2 Identification of communication channels

The requirement for segregation required that the communication channels between applications be made overt. This then allowed a decision as to whether such channels were appropriate, both in existence, and in bandwidth.

The *Scos* application provides some functionality to the user applications, including a random number generator. In order to faithfully model the way the generator works, it was necessary to introduce an overt communication channel. This exposed the fact that there is a potential communication of the random number between applications; further analysis demonstrated that this channel could not in fact be used to pass any useful information.

9.3 Proof detected an error

An early version of the design had a rather subtle error to do with clearing RAM when swapping between applications. In one very special case, which required unloading the only application on a card, then loading another in a special mode, the RAM was not properly cleared, resulting in a potential covert communication.

This error was detected both by the design team and by the formal modelling. Interestingly, it was not the proof effort itself that detected the error, it was in the mapping between the formal model and the semi-formal design. The *VM* model is deterministic, so it fully specifies the contents of RAM, and segregated, so it specifies the contents of RAM for each unpromoted application. The most sensible deterministic specification of the contents of RAM for a newly loaded application is to set it to some predefined 'cleared' value: this clearing did not occur in the semi-formal model, and so the mapping detected a flaw. (Had the formal model been written from the semi-formal model, the proof of determinism would not have been possible, which would have uncovered the flaw at that point.)

So, in this case, just *thinking* about what the proof obligations were going to be influenced how we wrote the model, and exposed the design flaw.

10 Lessons learned

10.1 Model structure versus proof structure

There is a fine balance between model structure clarity and ease of proof. In order to prove the difficult property of segregation, we sacrificed some clarity for

ease of proof (by using a free type to bundle the different kinds of applications into one, and by requiring the global promotion function to be total), and we sacrificed some ease of proof for clarity (by using the more conventional form of multi-promotion, and proving it equivalent to the unwound form).

We experimented with the alternative approaches before converging to this particular compromise.

10.2 Presentation

The specification is large, and the proof structure subtle. The third party reviewers had to navigate a complex document, had to be able to find definitions, and had to be assured nothing had been left out.

The index and the proof tree chapter were arguably two of the most important sections in the final document. Even the authors found these chapters essential when coming back to the document after a break!

10.3 Providing further justification

On their first pass through the model, the third party reviewers raised an observation about the size of the input space: we had formally modelled more input messages than could actually be implemented. The reviewers wanted us to justify that this was not a problem: that the implemented restriction could not be used to covertly signal information. We provided such a justification (as an appendix) in the next version of the model.

So the formal development process can be iterative: external comments can require rethink, further justifications, and more detail to be provided.

10.4 Elegant mathematical results may not help

Just because the statement of a proof obligation is simple and elegant, does not mean that its application to a particular problem will be simple and elegant. Much hard, potentially messy, proof work may be required.

We had an elegant formulation of segregation, but it was not in a form that mapped naturally to the conventional state-and-operations style of Z specification we used for the modelling work. Even after moving the result into the Z world, and unwinding it to a multi-promotion form, it still did not allow a natural specification style. So we did not use it in the modelling, which necessitated us discharging an extra proof obligation.

We also used a specification trick to define the HW model operations in terms of the VM model operations. This simplified the modelling enormously, and all we had to do to prove refinement was to prove that the retrieve was functional, total, and surjective [Woodcock & Davies 1996, section 18.3]. The proof obligation can be expressed in one line:

$$\vdash R \in HWState \nrightarrow VMState$$

However, that one line hides a wealth of messy and not very interesting detail: the state spaces of both states have many components. After expanding out the states and the retrieve, the mere statement of the proof obligation extends over several pages. The proof itself was quite cumbersome.

11 Summary

We have proved the correctness of the refinement, and the preservation of a security property, of a real industrial product, working to real development timescales. In the process, we uncovered a security flaw in one part of the system design (to do with clearing memory under some unusual conditions), and identified the corrections needed.

We achieved a very high level of rigour in our proofs. The proofs are far more detailed than typical proofs done in general mathematics. Despite this the formal methods activity was never on the critical path of the development. The formal methods component was usually ahead of schedule, and never caused a delay in development.

As a byproduct of doing these proofs, we have also generalised the notion of segregation to allow controlled communication, and applied it in a Z state-and-operations style.

Acknowledgements

The work described in the paper took place as part of a development for the NatWest Development Team.

Parts of the work were carried out by Eoin Mc Donnell, Barry Hearn and Andy Newton (all of Logica). We would like to thank Jeremy Jacob and John Clark for their helpful comments and careful review of this work.

References

[Barden *et al.* 1992]
Rosalind Barden, Susan Stepney, and David Cooper. The use of Z. In John E. Nicholls, editor, *Proceedings of the 6th Annual Z User Meeting, York 1991*, Workshops in Computing, pages 99–124. Springer Verlag, 1992.

[Barden *et al.* 1994]
Rosalind Barden, Susan Stepney, and David Cooper. *Z in Practice*. BCS Practitioners Series. Prentice Hall, 1994.

[Bell & Padula 1976]
David E. Bell and Len J. La Padula. Secure computer system: unified exposition and MULTICS. Report ESD-TR-75-306, The MITRE Corporation, March 1976.

[Bell 1988]
D. E. Bell. Concerning "modelling" of computer security. In *Proceedings 1988 IEEE Symposium on Security and Privacy*, pages 8–13. IEEE Computer Society Press, April 1988.

[Cooper & Stepney 2000]
David Cooper and Susan Stepney. Segregation with communication. (These proceedings), 2000.

[DSDM Consortium]
DSDM Consortium. Dynamic Systems Development Method manual. Technical report, `http://www.dsdm.org/`.

[Flynn *et al.* 1990]
Mike Flynn, Tim Hoverd, and David Brazier. Formaliser—an interactive support tool for Z. In John E. Nicholls, editor, *Z User Workshop: Proceedings of the 4th Annual Z User Meeting, Oxford 1989*, Workshops in Computing, pages 128–141. Springer Verlag, 1990.

[Goguen & Meseguer 1984]
J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proceedings 1984 IEEE Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society, 1984.

[Gollman 1998]
Dieter Gollman. *Computer Security*. John Wiley, 1998.

[Jacob 1992]
Jeremy L. Jacob. Basic theorems about security. *Journal of Computer Security*, 1(4):385–411, 1992.

[Josephs 1991]
Mark B. Josephs. Specifying reactive systems in Z. Technical Report TR-19-91, Programming Research Group, Oxford University Computing Laboratory, 1991.

[Roscoe 1995]
A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings 1995 IEEE Symposium on Security and Privacy*, pages 114–127. IEEE Computer Society Press, 1995.

[Rushby 1981]
J. M. Rushby. The design and verification of secure systems. In *Proceedings 8th ACM Symposium on Operating System Principles*, December 1981.

[Spivey 1992a]
J. Michael Spivey. *The fuzz Manual*. Computer Science Consultancy, 2nd edition, 1992. ftp://ftp.comlab.ox.ac.uk/pub/Zforum/fuzz.

[Spivey 1992b]
J. Michael Spivey. *The Z Notation: a Reference Manual*. Prentice Hall, 2nd edition, 1992.

[Stepney]
Susan Stepney. Formaliser Home Page. http://public.logica.com/~formaliser/.

[Stepney *et al.* 1998]
Susan Stepney, David Cooper, and Jim Woodcock. More powerful Z data refinement: pushing the state of the art in industrial refinement. In Jonathan P. Bowen, Andreas Fett, and Michael G. Hinchey, editors, *ZUM'98: 11th International Conference of Z Users, Berlin 1998*, volume 1493 of *Lecture Notes in Computer Science*, pages 284–307. Springer Verlag, 1998.

[Toyn 1996]
Ian Toyn. Formal reasoning in the Z notation using CADiZ. In N. A. Merriam, editor, *2nd International Workshop on User Interface Design for Theorem Proving Systems*. Department of Computer Science, University of York, July 1996. http://www.cs.york.ac.uk/~ian/cadiz/.

[Valentine *et al.* 2000]
Sam Valentine, Ian Toyn, Susan Stepney, and Steve King. Type constrained generics. (These proceedings), 2000.

[Woodcock & Davies 1996]
Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, 1996.

A Conjectures with *f*UZZ

We did all our proofs by hand, but we did use *f*UZZ to typecheck them, which provided a valuable level of tool support. However, *f*UZZ does not support the syntax for conjectures, so we had to make judicious use of its various **%%** directives, to allow the same markup to be both checked by *f*UZZ and typeset by LaTeX.

The semantics of conjectures are different from those of predicates, but the type rules are the same. So the technique we use it to present a conjecture to *f*UZZ as if it were a predicate, using **%%ignore** to hide the turnstile, and present it to LaTeX to typeset correctly, by using **%%** to hide the parts *f*UZZ needs but LaTeX does not.

Instruct *f*UZZ to ignore turnstiles.

```
%%ignore \shows
```

A.1 Non-generic conjectures

Markup a simple non-generic conjecture as a Predicate paragraph, hiding from LaTeX the predicate's $\forall$ and $\bullet$.

The markup

```
\begin{zed}
%%\forall
y:\nat
%%@
\\ \shows
\\  y=y
\end{zed}
```

appears to *f*UZZ as

```
\begin{zed}
\forall
y:\nat
@
\\
\\  y=y
\end{zed}
```

and so typechecks

appears to LaTeX as

```
\begin{zed}
y:\nat
\\ \shows
\\  y=y
\end{zed}
```

and so typesets as

$$y : \mathbb{N}$$
$$\vdash$$
$$y = y$$

A.2 Generic conjectures

Markup a generic conjecture as a Generic-Box paragraph, hiding from LaTeX the box markup (which means LaTeX needs an extra math-mode markup), the (dummy) declaration, and the predicate's $\forall$ and $\bullet$.

The markup

```
\[
%%\begin{gendef}
[X]
%% dummy42 : X
\\
%%\where
%%\forall
y:X
%%@
\\ \shows
\\ y=y
%%\end{gendef}
\]
```

appears to *f*uzz as

```
\begin{gendef}
[X]
dummy42 : X
\\
\where
\forall
y:X
@
\\
\\ y=y
\end{gendef}
```

and so typechecks

appears to LaTeX as

```
\[
[X]
\\
y:X
\\ \shows
\\ y=y
\]
```

and so typesets as

$$[X]$$
$$y : X$$
$$\vdash$$
$$y = y$$