Modelling Mobility Aspects of Security Policies

Pieter Hartel, Pascal van Eck, Sandro Etalle, and Roel Wieringa

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
{pieter,vaneck,etalle,roelw}@cs.utwente.nl

Abstract. Security policies are rules that constrain the behaviour of a system. Different, largely unrelated sets of rules typically govern the physical and logical worlds. However, increased hardware and software mobility forces us to consider those rules in an integrated fashion. We present SPIN models of four case studies where mobility plays a role. At present our models are ad-hoc. In each case the model captures both the system of interest and its security policy. The model is then formally checked against a security principle. The model checking activity shows examples of policies that are too weak to cope with mobility.

1 Introduction

Security policies are important both in the physical and the logical world. By a *policy* we mean "a rule that defines a choice in the behaviour of a system" [1]. A *security* policy rules out behaviour "that has been deemed unacceptable" [2]. A *spatial* security policy constrains this further by ruling out behaviour tied to particular locations [3]. Physical events may affect logical security and vice versa. For example a user moving a laptop with access to confidential data from a secure to an insecure environment must loose all authorisations on the data immediately. The problem we intend to solve is finding a means to analyse security policies from the point of view of logical and physical mobility. The urgency of solving this problem is caused by increased mobility. For example (small) mobile computing equipment is easily carried out of a building, past unsuspecting security guards, regardless of any measures like encryption, or tamper resistance. Mobility thus ties logical and physical security together, causing new and, as we will show, unanticipated security problems to arise.

We propose a first step towards an analysis method, with initial tool support. The method is based on developing a formal model of a system with its security policy, and on using a model checker to analyse the combination. From the analysis we are able to predict the occurrence of security problems that would not have occurred without mobility. We present case studies from four different domains to illustrate the method.

In our work, we do not distinguish between logical mobility (mobile code that roams from one processing and networking context to another) and physical mobility [4] (non-mobile code on mobile hardware that is carried from one network context to another by its user). In both cases, the execution context of some code under study changes, which is the level of abstraction at which we work.

Our method consists of manually creating (1) an abstract model of a system of interest, (2) its security policy and (3) a security design principle that the combination of (1) and (2) should satisfy.

The system model should be able to generate all possible behaviours of the system of interest, including behaviours of hacked versions of the system. For example consider the UNIX ping command, which normally makes a characteristic sequence of system calls. Hacked versions of ping would make different sequences of system calls.

The security policy should constrain the behaviours of the system model to desirable (safe) behaviours. For example, an execution monitoring system might allow ping to make a raw socket call (which requires root permission), but no other potentially dangerous systems calls.

The security principle finally constrains the joint behaviour of the system and its security policy to something the system designer would have had in mind when creating the system. In the case of ping this might be the principle of the least privilege, which would ensure that root permission is only used once, for the raw socket call.

We express our system, policy and principle in Promela, the input language of the SPIN model checker [5] as follows:

A System of interest is characterised by a trace of relevant events, for example system calls, or messages across a network. This is modelled in a most general way to capture only the essence of the system, particularly its mobility aspects. The system is modelled in SPIN by global data, channels and processes.

A Security Policy constrains the behaviour of the system to acceptable (safe) behaviour. The policy constrains traces of events using the same terms as the system, e.g. system calls, network messages etc. The policy is modelled by one or more processes.

A Security Principle is a design guideline for the system and the policy. To make the principle operational we translate it into a specification for the system and its policy, which, in our case studies, takes the form of a SPIN trace declaration (See [5, Page 485] for the technical reason why we use trace declarations rather than LTL formulae).

The system, policy and principle satisfy:

$$(\mathit{system} \parallel \mathit{policy}) \models \mathit{principle}$$

We use SPIN to analyse a system and policy with respect to a principle; a system and policy that does not abide by the principle gives rise to concrete counter examples. We demonstrate the approach by presenting four case studies in subsequent sections. In each case we identify system, policy, and principle. SPIN then shows that no deadlock can occur in ($\mathit{system} \parallel \mathit{policy}$). This shows that we are working with sensible models that admit behaviour satisfying the security policy. However, model checking (*system* $\|$ *policy*) $\models$ *principle* gives traces showing that the principle can be violated. In all but the first case study (which is intended as an introductory example) the violations can be attributed to mobility.

Our modelling methodology is based on a combination of techniques developed by Alan Mycroft and his group from Cambridge, UK, for dynamic security policies in the logical domain (security policy abstraction [6]) and the physical domain (spatial security policy [3]). We extend the Cambridge work in the logical domain [6] by adding mobility considerations. Both works from Cambridge propose policy languages but no realised tool support. The inspiration for using SPIN by way of tool support comes from the work of Cheng et al [7], who also use SPIN to model check security policies, however without consideration of either mobility or the physical domain. A third source of inspiration is formed by the work of Sekar et al [8], who present a system which automatically derives models from code. Model checking is then used to analyse models with respect to a choice of security policies. Sekar et al do not consider physical mobility.

The four subsequent sections discuss one case study each. The final section concludes and discusses further work.

2 Ping

The first case study concerns the UNIX utility ping, which sends an IP packet to a network node, reporting on the availability of the destination and on the performance of the connection. Making the connection requires root permission (for the socket call), which is potentially dangerous and should thus be minimised. This is captured by the principle of the least privilege, which states that [9]: "Every program and every user of the system should operate using the least set of privileges necessary to complete the job". In the case of ping (and other commands) the principle translates into *dropping root permission after executing the first socket system call.*

This case study shows that the translation of a general principle such as least privilege to the case at hand is relative to the design charter that is given a-priori. In this case study, we assume that the design of the ping utility is within the design charter, while the design of the UNIX operating system is not. Consequently, it is a given that some system calls needed in the ping utility need root permission, which results in the translation of the principle of least privilege as given above. If the design of UNIX itself would have been within the design charter of this case study, most probably a solution would have been chosen in which the ping utility would not need root permission.

2.1 The System

We model behaviour by traces of system calls, which is abstract because we ignore the parameters of the system calls and any associated calculations. At the same time, naming system calls is concrete because we distinguish system calls that do not need root permission, and which could be collapsed. The Promela enumeration type `mtype` below mentions the same system calls as used by Madhavapeddy et al [6].

```
mtype = {
   LibC_exit, LibC_gethostbyname, LibC_gettimeofday, LibC_printf, SysCall_brk,
   SysCall_mprotect, SysCall_recvfrom, SysCall_sendto, SysCall_sigaction, SysCall_socket
} ;
```

We use a synchronous channel to connect the ping system to the ping policy. A message consists of one of the symbols `LibC_exit` ... `SysCall_socket`.

```
chan c = [0] of {mtype} ;
```

The ping system is modelled abstractly by the process `ping_system`. The behaviour generated encompasses all possible traces of the ten system calls in the enumeration type `mtype`. This represents considerably more behaviour than real ping commands would exhibit and includes for example hacked versions of ping. This gives us a good model of reality.

```
active proctype ping_system() {
   do
   :: c!LibC_exit                         :: c!LibC_gethostbyname
   :: c!LibC_gettimeofday                 :: c!LibC_printf
   :: c!SysCall_brk                       :: c!SysCall_mprotect
   :: c!SysCall_recvfrom                  :: c!SysCall_sendto
   :: c!SysCall_sigaction                 :: c!SysCall_socket
   od
}
```

2.2 The Policy

The ping policy below represents an abstract version of the security policy described by Madhavapeddy et al [6]. (We have abstracted away from the fact that extra `brk` and `printf` system calls are always allowed.)

We define C pre-processor macros corresponding to the language constructs of the same name proposed by Madhavapeddy et al [6]. (The backward slashes at the end of each line except the last ensure that the macro definition extends across multiple lines.)

```
#define optional( x ) \                 | #define multiple( x ) \
   if \                                 |    do \
   :: x \                               |    :: x \
   :: skip \                            |    :: break \
   fi                                   |    od
```

The first non-deterministic choice below represents a call to ping that responds with a usage message. The second choice represents ping doing its proper work, i.e. a socket call, optionally followed by a call to `LibC_gethostbyname` etc.

```
active proctype ping_policy() {
   do
   :: c?LibC_printf ;
      c?LibC_exit
   :: c?SysCall_socket ;
      optional(c?LibC_gethostbyname) ;
      c?LibC_printf ;
      multiple(c?SysCall_sigaction) ;
      multiple(c?SysCall_sendto ; c?SysCall_recvfrom ; optional(c?SysCall_brk)) ;
   od
}
```

To understand how the ping system and the policy interact, compare the send actions `c!`... of `ping_system` to the receive actions `c?`... of `ping_policy`. This comparison reveals that the former is prepared to engage in any send action, whereas the latter is prepared only to engage in specific receive actions. This then explains how the policy constrains the generic behaviour of the system. We use the same technique throughout the paper to effectuate the security policies.

The separation of system and policy thus provides a convenient way to talk about the system (which is general) and the policy (which constrains general behaviour to specific behaviour). To indicate that this not an entirely trivial result we point out that many other ping implementations are possible that cannot be constrained to conform to the policy, for example a version of `ping_system` where the `do` ... `od` above would be replaced by `if` ... `fi`.

The security policy still permits attacks, for example mimicry attacks [10], which subtly alter the pattern of system calls to achieve malicious intent while avoiding detection by an IDS. Our use of a security principle below also captures this type of attack.

2.3 The Principle

The ping policy must satisfy the principle of the least privilege. This is mentioned but not explicitly specified as such by Madhavapeddy et al [6]. As stated before, the principle is interpreted as ping must drop root permission after one socket call, which we model here by saying that one socket call is ok, but not two. This is captured by the trace declaration below, which matches a trace that has exactly one `SysCall_socket`. When model checking, SPIN tries to find a sample trace that does not match the trace declaration, which is then a counter example for the desired principle.

```
#define anything_but_socket() \
      c?LibC_exit
   :: c?LibC_gethostbyname \
   :: c?LibC_gettimeofday
   :: c?LibC_printf \
   :: c?SysCall_brk
   :: c?SysCall_mprotect \
   :: c?SysCall_recvfrom
   :: c?SysCall_sendto \
   :: c?SysCall_sigaction
```

```
trace {
   do
   :: anything_but_socket()
   :: c?SysCall_socket ->      break
   od ;
   do
   :: anything_but_socket()
   od
}
```

2.4 Analysis

Model checking the parallel composition of `ping_system`, `mobility` as well as `ping_policy` against the built in formulae of the SPIN model checker (absence of deadlock) shows no errors. This demonstrates that the model is indeed a sensible one because behaviours are possible that satisfy the security policy.

Model checking the system and the policy against the principle reveals traces that do not match the trace declaration, i.e. traces that violate the principle. A concrete example is `SysCall_socket`; `LibC_printf`; `SysCall_socket`. To remedy the situation we have several options. For example we could replace `do` ... `od` in the ping policy by `if` ... `fi`, because then only one socket call would result.

A better alternative would be to exit the loop once the second non-deterministic choice has completed, because this allows an arbitrary number of 'usage' message to be generated but one 'proper' ping call.

The results seem trivial but we should like to point out that our system model and security policy are indeed representative for current intrusion detection systems. Therefore it is encouraging that SPIN has been able to discover a problem with our system and policy.

3 Database Application

The second case study investigates the separation of testing and production environments in a modern central database application. The application consists of three layers: data, business logic and presentation. The data and business logic layers reside on a central server. There are two datasets: one with production data and one with randomised test data. The business logic layer accepts network connections from the presentation layer, which consists of Java applets available throughout the organisation. Upon accepting a connection from a presentation layer client, the business logic layer should check the physical location of the client in a configuration database and depending on this location, use either the production or test dataset. We treat the presentation layer and the data base layer as the system, and the business logic layer as the policy because the business logic layer decides what constitutes acceptable behaviour.

3.1 The System

We define the symbols necessary for our example. `Test_0` represents the test data, `Data_0` ... `Data_3` represent production data. The symbols `Test_Env` and `Production_Env` identify the test and production environments, and `Connect` ... `Reply` represent commands exchanged between the three layers of the system.

```
mtype = {
   Test_0, Data_1, Data_2, Data_3, Test_Env, Production_Env,
   Connect, Disconnect, Request, Reply
}
```

The model comprises three processes (representing the presentation, business logic and database layers) and two synchronous channels connecting the layers. `p2b` connects the presentation to the business logic layer and `b2d` connects the business logic layer to the database layer. The message header may be `Connect` ... `Reply`. Depending on the message header, the message body may be one of `Test_0` ... `Data_3` or `Test_Env` or `Production_Env`.

```
chan p2b = [0] of {mtype, mtype} ;
chan b2d = [0] of {mtype, mtype} ;
```

The presentation layer process below generates an almost arbitrary sequence of requests for both production and test environments. The only form of protocol obeyed is that the presentation layer insists that it receives a reply after each request. The database layer reports test or production data as appropriate.

```
active proctype presentation_layer() {
  mtype data ;
end:
   do
   :: p2b!Connect(Test_Env)
   :: p2b!Connect(Production_Env)
   :: p2b!Request(Test_Env) ->
      p2b?Reply(data)
   :: p2b!Request(Production_Env) ->
      p2b?Reply(data)
   :: p2b!Disconnect(Test_Env)
   :: p2b!Disconnect(Production_Env)
   od
}
```

```
active proctype database_layer() {
   mtype data ;
end:
   do
   :: b2d?Request(Production_Env) ->
      if
      :: data = Data_1 ;
      :: data = Data_2 ;
      :: data = Data_3
      fi ;
      b2d!Reply(data)
   :: b2d?Request(Test_Env) ->
      b2d!Reply(Test_0)
   od
}
```

3.2 The Policy

The business logic layer process mediates between the presentation and the database layers, ensuring behaviour consistent with the business rules. In particular the business logic layer insists that applications make data base requests only when connected to the data base. Each request is passed on to the database layer, indicating whether to use the test or production data base. (The function `eval` ensures that the body of the `Reply` message matches exactly with the current value of `data`).

```
active proctype business_logic_layer() {
   mtype env, data ;
end:
   do
   :: p2b?Connect(env) ->
      do
      :: p2b?Request(env) -> b2d!Request(env) ; b2d?Reply(eval(data)) ; p2b!Reply(data)
      :: p2b?Disconnect(env) -> break
      od
   od
}
```

3.3 The Principle

The principle of interest is that "Testing and production environments are physically separated". This is enforced by checking that once connected from a particular environment, all following request messages emanate from that same environment until a disconnect message arrives, again from the same environment. An implementation that checks the location once for each connection suffices in the case of a fixed network. In the case of a mobile network that supports roaming, this does not suffice.

```
trace {
   do
   :: p2b?Connect(Test_Env) ->
      do
      :: p2b?Request(Test_Env) ->        p2b?Reply(_)
      :: p2b?Disconnect(Test_Env) ->     break
      od
   :: p2b?Connect(Production_Env) ->
      do
      :: p2b?Request(Production_Env) ->    p2b?Reply(_)
      :: p2b?Disconnect(Production_Env) -> break
      od
   od
}
```

3.4 Analysis

Model checking the system and the policy against the principle reveals (as expected) that the presentation layer is ill behaved because message sequences with arbitrary test and production environment parameters are generated. The following trace gives a concrete counter example for our principle: `Connect(Test_Env)`, `Request(Production_Env)`.

The problem lies in the business logic layer (the policy), which should constrain the presentation layer to correct behaviour. Incorporating the principle in the form of a trace declaration points out this deficiency of the business logic layer. The business logic layer might implement the principle using `eval(env)` instead of plain `env` in the receive actions for `Request` and `Disconnect`. This would constrain the environment of the call to match the value of the `env` variable exactly.

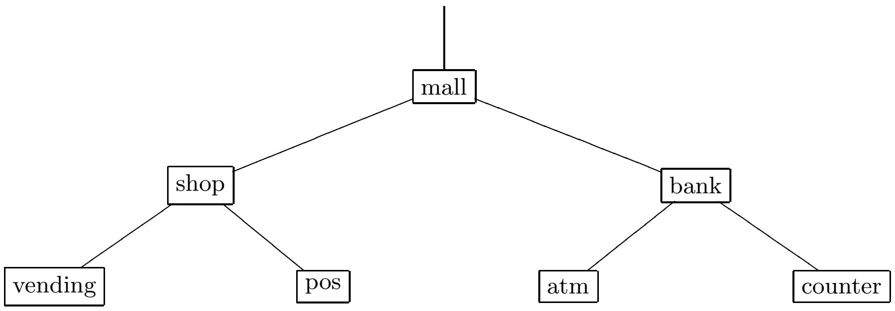

Fig. 1. Mall with a vending machine and a point of sale terminal (pos) at the shop, and an automated teller machine (atm) and the traditional counter at the bank.

4 Smart Cards

The third case study investigates the behaviour of next generation smart cards, which are the object of study in the European Inspired project[1]. The principle of interest here is *Be reluctant to trust*. To operationalise this principle we propose security policies that can be customised by the card issuer as well as the card holder. The concrete example that we will study is of (1) a card holder who states that she *does not permit applets to be loaded (i.e. smart card management)*

[1] http://www.inspiredproject.com

other than when she is at the bank and (2) a card issuer who states that *a payment transaction at a vending machine must always be followed by a loyalty transaction.* Using a smart card that operates under the Be reluctant to trust principle would give the user trust in the system because the user can hold the bank responsible for all the applets on the card. This would also mean that a shop could not load a loyalty applet unless the permission to load applets is explicitly delegated to the shop.

4.1 The System

Following Madhavapeddy et al [6] we assume a tree-shaped world model. Figure 1 shows a shopping mall at the root of the tree, a shop and a bank are at the intermediate layer, and four smart card readers at the leaves.

We use synchronous channels to model the identity of the (physically separated) parties. Three channels connect to the physical locations, four to the smart card readers and another three to the applets. The last three are shown here, the former are defined in the macros `run_leaf` and `run_node` below.

```
chan loyalty = [0] of {chan} ;
chan payment = [0] of {chan} ;
chan management = [0] of {chan} ;
```

We need two kinds of processes: one type representing interior nodes (embedded in macro `run_node`) and one for leaf nodes (macro `run_leaf`). (The macros generate both a channel with the given name, for example `vending`, and corresponding process `proc_vending`.) To model mobility, each process allows an applet to travel from any of its input channels to any of its output channels. In the node process this means that an applet can be received either from the parent of the node, or from one of the children. Similarly, the applet can be moved on to the parent or one of the children.

```
#define run_node(parent, left, right) \
chan parent = [0] of {chan} ; \
active proctype proc_##parent() { \
   chan applet ; \
end: \
   do \
   :: if \
      :: parent?applet   :: left?applet   :: right?applet \
      fi ; \
      if \
      :: parent!applet   :: left!applet   :: right!applet \
      fi \
   od \
}
```

A leaf process can exchange applets with the parent only. An applet can in principle be executed on a smart card connected to a smart card reader located at the leaf. This is modelled by sending the identity of the node onto the applet channel thus: `applet!parent`. A leaf makes a non-deterministic choice whether to execute the applet or not.

```
#define run_leaf(parent) \
chan parent = [0] of {chan} ; \
active proctype proc/**/parent() { \
   chan applet ; \
end: \
   do \
   :: parent?applet ; \
      if \
      :: applet!parent \
      :: skip \
      fi ; \
      parent!applet \
   od \
}
```

The world is instantiated to the configuration shown in Figure 1 by the seven macro calls below.

```
run_leaf(vending)             run_leaf(pos)
run_node(shop, vending, pos)
run_leaf(atm)                 run_leaf(counter)
run_node(bank, atm, counter)  run_node(mall, shop, bank)
```

The system described above is general. It allows free travel of applets, and is prepared to interact with any applet at any of the nodes. This is too liberal, and we need a policy to constrain the resulting behaviour to acceptable behaviour.

4.2 The Policy

The init process represents the policy that constrains the behaviour of the system. The init process begins by injecting a loyalty applet, a payment applet and a management applet into the system (via the parent channel of the root node `mall`). Eventually the mall will return the three applets, whence the system terminates. During its life time, the init process is prepared to receive the identity of the host of any of the applets on the corresponding channels `payment`, `loyalty`, and `management`, indicating that the relevant applet is executed while the smart card is connected to the indicated reader.

```
init {
   chan host ;
   mall!loyalty ;
   mall!payment ;
   mall!management ;
end:
   do
   :: loyalty?host             :: payment?host
   :: management?host          :: mall?eval(loyalty)
   :: mall?eval(payment)       :: mall?eval(management)
  od
}
```

4.3 The Principle

The trace specification below represents the principle *Be reluctant to trust* as operationalised by the two customised policies: one for the smart card issuer and one for the smart card holder. The first non-deterministic choice below represents a payment transaction at the vending machine that must be followed by a loyalty transaction at the vending machine. The second non-deterministic choice represents that the only possibility for a management transaction to take place is at the counter of the bank. The other non-deterministic choices represent the remaining desirable behaviour.

```
trace {
end:
  do
  :: payment?eval(vending) ;loyalty?eval(vending)
  :: management?eval(counter)                          :: payment?eval(pos)
  :: payment?eval(atm)                                 :: payment?eval(counter)
  :: loyalty?eval(vending)                             :: loyalty?eval(pos)
  :: loyalty?eval(atm)                                 :: loyalty?eval(counter)
  od
}
```

4.4 Analysis

Model checking reveals (again as expected) that the system is ill behaved because the applets roam freely and therefore execute when the principle prohibits this. A concrete counter example shows that after some preliminaries the management applet travels to the point of sale terminal thus: `mall!management`, `shop!management`, `pos!management`. There the management applet executes, which is modelled by sending the identity of the host back to the init process: `management!pos`. The violation of the card holder specific part of the principle can be prevented in the policy by replacing `:: management?host` by `:: management?eval(counter)`. It is not easy to enforce also the card issuer specific part of the principle as this links two events (the payment and the loyalty transaction) that could in principle be separated by an arbitrary number of unrelated events. To introduce such linkage into the policy, a notion of history would have to be included, for example by adding a variable to the model. This shows that the separation of principle and policy brings at least notational convenience that would not be available otherwise.

5 Peer to Peer Music Sharing

The last case study concerns a peer to peer music sharing system [11]. The model of the relevant processes, message flows and computations is given in Figure 2. The boxes denote processes and their internal actions, the arrows denote messages exchanged between the processes. We will discuss each of the processes and messages below. The model is abstract in the sense that:

– We assume that there are two different classes of users: music (1) producers and (2) consumers; there are valid and expired (3) leases; there are valid and invalid (4) payment tokens; there is (5) music in plain text form, (6) encrypted music, and (7) watermarked music; there are (8) keys and there are (9) fingerprints. All of the above 9 categories are assumed to be distinct and incompatible, for example a fingerprint cannot be confused with the identification of music.

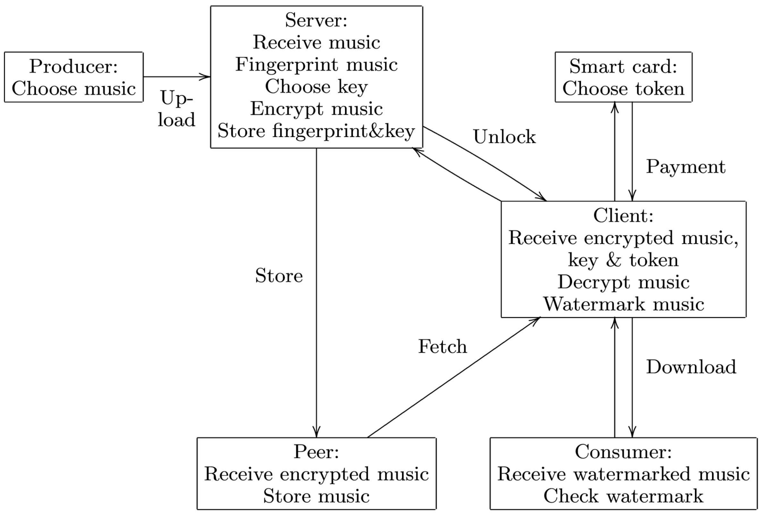

Fig. 2. Abstract Music2Share Protocols.

- We assume: (a) a simplistic peer network where peers do not actively redistribute or copy content; (b) the existence of a secure mechanism for looking up the fingerprint for the desired music; (c) idealised encryption, fingerprinting and watermarking.
- We assume that the producer and the server form a secure domain, that the client and the smartcard form another secure domain, and finally that the communication between the client and the server is secure. We make no security assumptions about the peers.
- Without loss of generality, we distinguish precisely two users, two pieces of music, two lengths of lease, two keys etc. This could be extended but no significant new lessons would be learned from doing so.
- We model small numbers of the different parties of the protocol, except the Server, which is centralised. Distributing the server could be accomplished but this would be a refinement that should remain invisible at the chosen level of abstraction.
- We assume synchronous communication so that the network does not have to store messages. We also assume the network to be reliable.

Under the assumptions above we are now able to present the two main scenarios of use: uploading and downloading music.

Scenario 1: upload. Starting top left in Figure 2, the producer chooses some music and uploads it onto the server (Upload message). The server receives the music, and calculates the fingerprint that will henceforth identify the music. The server then chooses an encryption key, and encrypts the music. The key is stored with the fingerprint for future use. An appropriate peer stores the encrypted music (Store message) for future reference. At some point in time the server decides that the lease of the music expires and invalidates the key (not shown in the diagram).

Scenario 2a: successful download. Starting bottom right in Figure 2, the consumer chooses some music (identified by its fingerprint) and requests the music from a client (Download request). We assume the client receives a valid token from the smartcard by way of payment (Token message). The client then asks the server for the key (Unlock request). We also assume that the lease has not expired so that the client receives a valid key (Unlock reply). The client also receives the encrypted music from the P2P network (Fetch message). The music can now be decrypted and watermarked with the identity of the consumer. The result is sent back to the consumer (Download reply).

Scenario 2b: failed download. The scenario will change if either payment could not be arranged (because the valid tokens of the smart card ran out), or when the lease has expired. In both cases the consumer receives an appropriate apology, but no music.

5.1 The System

The relevant symbols of the model are:

```
mtype {
   Download, Fetch, Store, Unlock, Payment, Upload,
   Alice, Bob, Alice_Music, Bob_Music, No_Music,
   Long_Lease, Short_Lease, Expired_Lease, Valid_Token, Invalid_Token,
   Bach, Mozart, Bach_Cipher, Mozart_Cipher, Bach_Key, Mozart_Key,
   Bach_Fingerprint, Mozart_Fingerprint
}
```

Here `Download` . . . `Upload` represent the different messages that may be transmitted; `Alice`, and `Bob` are the users wishing to download music; `Alice_Music`, and `Bob_Music` represent music watermarked by the identity of the user who downloaded the music; `No_Music` is a place holder for music whose lease has expired; `Long_Lease` . . . `Expired_Lease` represents three different lengths of lease for shared music; `Valid_Token`, and `Invalid_Token` represent two possible payment token values; `Bach`, and `Mozart` represent two pieces of music; `Bach_Cipher`, and `Mozart_Cipher` represent the encrypted versions of the two pieces of music; `Bach_Key`, and `Mozart_Key` represent the encryption keys for the two pieces of music; `Bach_Fingerprint`, and `Mozart_Fingerprint` represent the fingerprints of the two pieces of music.

For technical reasons, we need a macro `ord` (below) to map `Mozart_Fingerprint` to 0 and `Bach_Fingerprint` to 1. `NIL` represents an out-of-band (error) symbol.

```
#define ord(m) (m - Mozart_Fingerprint)
#define NIL    0
```

Keys. We assume a unique key for each piece of music. (In addition to C pre-processor macros, SPIN also offers its own variety of macro, the `inline`):

```
inline choose_key(plain, key) {
   if
   :: plain==Bach   -> key=Bach_Key
   :: plain==Mozart -> key=Mozart_Key
   fi
}
```

Given a `plain` text, `choose_key` returns the corresponding `key`.

Fingerprinting and watermarking. Similarly for each piece of music there is a unique fingerprint:

```
inline fingerprint(plain, id) {
   if
   :: plain==Bach   -> id=Bach_Fingerprint
   :: plain==Mozart -> id=Mozart_Fingerprint
   fi
}
```

Once the user has downloaded her Bach or her Mozart, the music is watermarked for her personal use. We are no longer interested in the particular piece of music, only in the user for whom it has been watermarked. An invalid plain text is returned as an invalid marked result `No_Music`.

```
inline watermark(plain, user, marked) {
   if
   :: plain!=NIL && user==Alice -> marked=Alice_Music
   :: plain!=NIL && user==Bob   -> marked=Bob_Music
   :: else                      -> marked=No_Music
   fi
}
```

It should be possible to check whether a piece of content has been watermarked with the correct user identity. `No_Music` does not have a watermark. An incorrect watermark causes a SPIN assertion to fail:

```
inline check_watermark(user, marked) {
   if
   :: user==Alice && marked==Alice_Music -> skip
   :: user==Bob && marked==Bob_Music     -> skip
   :: marked==No_Music                   -> skip
   :: else                               -> assert(false)
   fi
}
```

Encryption and decryption. For each piece of music there is one cipher text:

```
inline encrypt(plain, key, cipher) {
   if
   :: plain==Bach && key==Bach_Key     -> cipher=Bach_Cipher
   :: plain==Mozart && key==Mozart_Key -> cipher=Mozart_Cipher
   fi
}
```

With a valid key, the cipher text decrypts uniquely to the original plain text. With an invalid (expired) key, the result is `NIL`:

```
inline decrypt(cipher, key, plain) {
   if
   :: cipher==Bach_Cipher && key==Bach_Key     -> plain=Bach
   :: cipher==Mozart_Cipher && key==Mozart_Key -> plain=Mozart
   :: else                                     -> plain=NIL
   fi
}
```

Network. The Music2Share network is modelled using two synchronous channels: one for `request` messages and another for `reply` messages. (Channels in SPIN are bi-directional). The channels carry messages with two parameters, where the message header is always one of `Download` ... `Upload`.

```
chan request=[0] of { mtype, mtype, mtype }
chan reply=[0] of { mtype, mtype, mtype }
```

Server. The key server is the only centralised component. The server must create and store keys, it must fingerprint and encrypt music, locate a suitable peer to store encrypted music, and it must serve keys.

Keys are stored together with a lease, which is decremented each time a key is served. This models the process of lease expiry. The data type declaration below defines type `record` holding a key and a lease.

```
typedef record {
   mtype key ;
   mtype lease ;
}
```

The `Server` itself sits in an endless loop waiting for one of two types of messages `Upload` and `Unlock` on the `request` channel.

```
proctype Server() {
   mtype cipher, id, key, plain, user ;
   record store[2] ;
   byte lease ;
   do
   :: request?Upload(plain, lease) ->
      fingerprint(plain, id) ;
      choose_key(plain, key) ;
      store[ord(id)].key = key ;    store[ord(id)].lease = lease ;
      encrypt(plain, key, cipher) ;
      request!Store(id, cipher)
   :: request?Unlock(id, user) ->
      if
      :: store[ord(id)].lease > Expired_Lease ->
         store[ord(id)].lease-- ;
         key = store[ord(id)].key
      :: else ->
         key = NIL
      fi ;
      reply!Unlock(key, user)
   od
}
```

Upon receipt of an `Upload` message with given `plain` text and `lease`, the server calculates the fingerprint `id`, chooses a `key`, stores the key and the lease in at the appropriate entry `ord(id)` in the array `store`, encrypts the plaintext with the key yielding a `cipher`, and finally transmits a `Store` request onto the network, expecting an appropriate peer to pickup the request and to store the cipher text. This completes the handling of the upload request, no acknowledgement is returned to the requestor of the upload (The network is assumed to be reliable at the chosen level of abstraction).

An `Unlock` request message with a fingerprint `id` and identity `user` causes the server to check the expiry of the lease for the music with the fingerprint `id`. If the lease has expired an invalid key (`NIL`) is created, otherwise the lease is shortened and the correct key retrieved. The key is posted on the `reply` channel, expecting the requestor of the unlock message to pick it up.

Peer. A peer is a simple process that serves only to store and communicate the cipher text corresponding to a particular fingerprint.

```
proctype Peer(mtype id) {
   mtype cipher ;
   request?Store(eval(id), cipher) ;
   do
   :: request?Store(eval(id), cipher)
   :: request!Fetch(id, cipher)
   od
}
```

Before the peer enters its main loop, it expects a `Store` message with the initial cipher text. (The expression `eval(id)` states that the actual parameter of the message must have exactly the same value as the variable `id`; the variable `cipher` on the other hand will be bound to what ever actual value is offered by an incoming message). In the main loop, the peer either offers the cipher text to a `Client` in need of the cipher text, or is ready to receive an updated cipher text. If a second process is waiting for a `request!Store` and a third process is waiting for a `request?Fetch`, both transactions are enabled. In this case a non-deterministic choice is made as to which transaction proceeds first.

Smart card. The Smart card represents a source of pre-paid tokens.

```
proctype Smartcard() {
   do
   :: request?Payment(_, _) ->
      if
      :: reply!Payment(Valid_Token, NIL)     :: reply!Payment(Invalid_Token, NIL)
      fi
   od
}
```

The tokens may run out, which is modelled by the `Invalid_Token`. Subsequent valid tokens are the result of recharging the card (not explicitly modelled).

Client. The Client mediates between the consumer of music and the Music2Share system.

```
proctype Client() {
   mtype cipher, id, key, plain, marked, user ;
   do
   :: request?Download(id, user) ->
      request!Payment(NIL, NIL) ;
      if
      :: reply?Payment(Valid_Token, _) ;
         request?Fetch(eval(id), cipher) ;
         request!Unlock(id, user) ;
         reply?Unlock(key, eval(user)) ;
         decrypt(cipher, key, plain) ;
         watermark(plain, user, marked) ;
         reply!Download(marked, user)
      :: reply?Payment(Invalid_Token, _) ->
         reply!Download(No_Music, user)
      fi
   od
}
```

The `Client` sits in an endless loop waiting for `Download` messages for a given fingerprint `id` and `user`. The first action is to check payment. If unsuccessful a `NIL` result is returned. Otherwise we `Fetch` the appropriate `cipher` text from a `Peer`. (No request message is necessary here as the peers offer cipher text unsolicited.) Then the client requests the key for the content. The reply message is matched to the identity of the user. After decryption and watermarking the `marked` music is returned to the consumer who posted the download request.

The client receives an invalid key if the lease is expired. In this case the marked result will also be `NIL`.

5.2 The Policy

The producer and the consumer form the endpoints in the value chain, and as such decide the policy for acceptable behaviour. The producer's policy is to upload a choice of music; the consumer downloads a choice of music.

```
proctype Producer() {
   do
   :: request!Upload(Mozart, Long_Lease)   :: request!Upload(Bach, Short_Lease)
   od
}
```

The `Producer` repeatedly tries to upload `Mozart` (on a long lease) and `Bach`, on a short lease. Further combinations could be added freely.

```
proctype Consumer(mtype user) {
   mtype marked ;
   do
   :: request!Download(Bach_Fingerprint, user) ->
      reply?Download(marked, eval(user)) ;
      check_watermark(user, marked)
   :: request!Download(Mozart_Fingerprint, user) ->
      reply?Download(marked, eval(user)) ;
      check_watermark(user, marked)
   od
}
```

The `Consumer` does the opposite of the producer: the consumer tries to `Download` content, checking that the downloaded content is indeed for the intended user. The content is identified by its fingerprint; we assume but do not model here the existence of a secure mechanism for looking up the fingerprint for the desired music.

Initialisation. The initialisation takes care that all processes are started with the appropriate parameters. Here we choose non-deterministically whether to use Alice or Bob as the consumer.

```
init {
   atomic {
      run Server() ;
      run Peer(Bach_Fingerprint)     ; run Peer(Mozart_Fingerprint) ;
      run Smartcard() ; run Client() ; run Producer() ;
      if
      :: run Consumer(Alice)         :: run Consumer(Bob)
      fi
   }
}
```

There is one `Server`, for all other processes we assume that at most two versions exist.

5.3 The Principle

The system policy states that she gets the music she has asked for, unless the lease expires, or she fails to pay. This is captured by the `check_watermark` assertion: a failed assertion implies that the policy is violated. This represents acceptable behaviour (from the point of view of the producer) but not desirable behaviour (because the consumer does not get value for money). The guiding principle is thus *value for money*, translated into a trace declaration requiring on the one hand that each time a consumers pays, he or she is guaranteed to get music, and on the other hand that when the customer cannot pay, she gets no music.

```
trace {
   do
   :: reply?Payment(Valid_Token, _) ;
      reply?Unlock(_, _) ;
      if
      :: reply?Download(Alice_Music, _)      :: reply?Download(Bob_Music, _)
      fi
   :: reply?Payment(Invalid_Token, _) ;
      reply?Download(No_Music, _)
   od
}
```

5.4 Analysis

Model checking the system and the policy reveals that the system does not cause failed assertions, showing that the policy is satisfied. However, the principle may be violated, because the server refuses to deliver an appropriate key once the lease on the music expires. A concrete trace is a little too long to show here; suffice to say that payment takes place, before we request the key. So if the lease expires and no key is forthcoming the user does not get value for money. Swapping the order of payment and key delivery would solve the problem, but at the same time we might introduce a new problem, whereby a key gets delivered for which payment may not be forthcoming. Further study is needed to identify a suitable business policy which would help to decide which alternative is preferred. The point here is that our methodology causes the right questions to be asked during the design stage.

A further point to note is that the trusted computing base (TCB) of the system is small: the producer, client, server and smart card must be secure, but the peers and the consumers do not have to be secure. The peers and the traffic to and from the peers is encrypted by the protocol, and may thus be transported freely on an open network. The music received by the consumer is watermarked with her identity, so that she can play and copy it for her own use, but if she tries to sell it, the watermark will reveal her identity.

6 Conclusions

We are able to model systems, security policies and security principles formally thus: $(\mathit{system} \parallel \mathit{policy}) \models \mathit{principle}$.

We have applied this idea to four case studies, showing how unexpected security problems arise that violate the principle.

In each of the four case studies the system is abstract, the policy is involved but the principle is short and clear. The systems and policies are more difficult to understand because of the concurrency involved. The principles by contrast are not concurrent.

None of the systems satisfy the relevant principle because of the mobility, showing that model checking leads to insight in the case studies.

SPIN's trace declarations are relatively inflexible. It would be useful to either increase the flexibility by changing the SPIN implementation, by generating trace declarations from higher level policies, or a combination of the two approaches.

We are planning to work on a language for policy patterns based on e.g. Ponder [1], from which SPIN models can be generated automatically. This would make it easier for practitioners to use our method. A particular challenge is to relate counter examples generated by the model checker back to the input language.

Finally we should like to investigate ways in which our models of policies and principles can be incorporated in applications by way of execution monitoring.

Acknowledgements

We thank Peter Ryan and the anonymous referees for their comments on an earlier draft of this paper.

References

1. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The ponder policy specification language. In Sloman, M., Lobo, J., Lupu, E., eds.: Int. Workshop on Policies for Distributed Systems and Networks (POLICY). Volume LNCS 1995., Bristol, UK, Springer-Verlag, Berlin (2001) 18–38 `http://springerlink.metapress.com/link.asp?id=1r0vn5hfxk6dxebb`.

2. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security **3** (2000) 30–50
`http://doi.acm.org/10.1145/353323.353382`.
3. Scott, D., Beresford, A., Mycroft, A.: Spatial security policies for mobile agents in a sentient computing environment. In Pezzè, M., ed.: 6th Fundamental Approaches to Software Engineering (FASE). Volume LNCS 2621., Warsaw, Poland, Springer-Verlag, Berlin (2003) 102–117
`http://www.springerlink.com/link.asp?id=nyxyyrlkbe5c5acc`.
4. Satoh, I.: Physical mobility and logical mobility in ubiquitous computing environments. In Suri, N., ed.: 6th Int. Conf. on Mobile Agents (MA). Volume LNCS 2535., Barcelona, Spain, Springer-Verlag, Berlin (2002) 186–201
`http://springerlink.metapress.com/link.asp?id=yrwh37hleb9rxp81`.
5. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference manual. Pearson Education Inc, Boston Massachusetts (2004)
6. Madhavapeddy, A., Mycroft, A., Scott, D., Sharp, R.: The case for abstracting security policies. In Arabnia, H.R., Mun, Y., eds.: Int. Conf. on Security and Management (SAM). Volume 1., Las Vegas, Nevada, CSREA Press (2003) 156-160
`http://cambridgeweb.cambridge.intel-research.net/people/rsharp/publications/sam03-secpol.pdf`.
7. Cheng, B.H.C., Konrad, S., Campbell, L.A., Wassermann, R.: Using security patterns to model and analyze security requirements. In Hietmeyer, C., Mead, N., eds.: Int. Workshop on Requirements for High Assurance Systems (RHAS), Monterey, California, Software Engineering Institute, Carnegi mellon Univ. (2003) 13–22
`http://www.sei.cmu.edu/community/rhas-workshop/rhas03-proceedings.pdf`.
8. Sekar, R., Venkatakrishnan, V.N., Basu, S., Bhatkar, S., DuVarney, D.C.: Model carrying code: A practical approach for safe execution of untrusted applications. In: 19th ACM Symp. on Operating Systems Principles (SOSP), Bolton Landing, New York, ACM Press, New York (2003) 15–28
`http://doi.acm.org/10.1145/945445.945448`.
9. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proceedings of the IEEE **63** (1975) 1278–1308
10. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: 9th ACM Conf. on Computer and communications security (CCS), Washington, DC, USA, ACM Press, New York (2002) 255–264
`http://doi.acm.org/10.1145/586110.586145`.
11. Kalker, T., Epema, D.H.J., Hartel, P.H., Lagendijk, R.L., van Steen, M.: Music2Share - Copyright-Compliant music sharing in P2P systems (invited paper). Proceedings of the IEEE Special Issue on Digital Rights Management **92** (2004) 961–970 `http://ieeexplore.ieee.org/xpl/abs_free.jsp?arNumber=1299170`.