Praxis
High Integrity
Systems

Document Set	Tokeneer ID Station		Reference S.P1229.41.2
Title	**:**	Formal Specification	
Synopsis	**:**	This document is the formal specification of the core Token ID Station (TIS), which forms part of Tokeneer.	
Contents	**:**	See table of contents	
Status	**:**	Definitive	
Issue Number	**:**	1.4	
Date	**:**	14th August 2008	
Copied To	**:**	**NSA** Randolph Johnson **SPRE Inc.**	**Praxis High Integrity Systems** Project Team
Front Sheet	**:**	Quality	
Originators	**:**	Janet Barnes	**Signed** **:**
Approver	**:**	David Cooper	**Approved** **:**

0 DOCUMENT CONTROL

Changes History

All issues of this document have been type-checked with fuzz *and have given no errors.*

Issue 0.1 (28th March 2003) First draft issued for comments within Praxis.

Issue 0.2 (5th April 2003) Draft incorporating comments from David Cooper.

Issue 0.3 (14th April 2003) Draft sent to Randolph Johnson (NSA).

Issue 0.4 (8th May 2003) Final Draft for formal review.

Issue 1.0 (9th May 2003) Provisional issue to client following internal review.

Issue 1.1 (27th June 2003) Definitive issue incorporating client feedback and correction to faults S.P1229.6.1-5.

Issue 1.2 (22nd July 2003) Definitive issue correcting faults

- S.P1229.6.6 - Tearing a user token mid-entry should constitute a failed entry and be logged as such within the statistics.
- S.P1229.6.7 - Remove the ability to retry fingerprint validation following a failure.
- S.P1229.6.19 - Correct typographical error on page 51.

Issue 1.3 (22nd August 2003) Definitive issue correcting faults.

- S.P1229.6.32 - Improve poor text messages on screen.
- S.P1229.6.33 - Make initial configuration realistic.
- S.P1229.6.36 - Screen should show busy message when a user entry is in progress.
- S.P1229.6.38 - Operation failures not reported on screen.

Issue 1.4 (14th August 2008) Updated for public release.

Changes Forecast

None. This document is now under change control.

References

1 The Z Notation: A Reference Manual, J.M Spivey, Prentice Hall, Second Edition, 1992

2 TIS Software Requirements Specification, Version 2.0, S.P1229.41.1.

3 TIS Kernel Protection Profile, SPRE Inc, Version 1.0, 5 February 2003.

4 TIS Security Target, S.P1229.40.1.

Abbreviations

AA	Attribute Authority
ATR	Answer-to-Reset
CA	Certification Authority
I&A	Identification and Authentication
RSA	Rivest Shamir Adelman algorithm
SPARK	SPADE Ada Kernel (analysable Ada subset from Praxis)
SRS	Software Requirements Specification
TIS	Token ID Station

1 TABLE OF CONTENTS

2 INTRODUCTION

In order to demonstrate that developing highly secure systems to the level of rigour required by the higher assurance levels of the Common Criteria is possible, the NSA has asked Praxis High Integrity Systems to undertake a research project to re-develop part of an existing secure system (the Tokeneer System) in accordance with their high-integrity development process. This re-development work will then be used to show the security community that is is possible to develop secure systems rigorously in a cost effective manner.

This document is the formal specification, written using the Z notation. This document specifies the behaviour of the core of the Token ID Station (TIS) that is being re-developed. It documents the second step in the Praxis high integrity systems development approach. The whole process consists of:

1. Requirements Analysis (the REVEAL process)
2. **Formal Specification (using the formal notation Z)**
3. Design (the INFORMED process)
4. Implementation in SPARK Ada
5. Verification (using the SPARK Examiner toolset).

2.1 Structure of this Specification

This specification is a formal model of the TIS core function presented using the Z notation. The specification models TIS as a number of state components and a number of operations that change the state. The operations presented in this specification cover:

- user authentication and entry into the enclave;
- enrolment of TIS;
- administrator logon/logoff;
- archiving the log;
- updating of configuration data;
- shutdown;
- overriding the enclave door.

This specification specifically does not model user exit from the enclave; there could also be further administrative operations above and beyond those presented in this specification but these are not considered. It is intended that the structure of the specification should not preclude the addition of further administrative operations.

The specification is structured by presenting type constructs useful in the modelling of TIS in the remainder of this section.

Section 3 introduces the state that defines the TIS.

Section 4 covers accepting data from the real world and updating the real world.

Section 5 presents a number of partial operations on parts of the TIS state, these are later used in the construction of the TIS system operations.

Section 6 presents the multi-phase user authentication and entry operation.

Section 7 describes all the system operations that take place within the enclave. These are administrative operations.

Section 8 defines the initial system and the state of TIS at start-up.

Section 9 describes how the whole TIS core works. Here we pull together the operations described through the remainder of the specification.

Appendix A gives a brief introduction to reading Z.

Appendix B discusses a number of issues that were raised during the production of this specification.

Appendix C gives an informal justification of the precondition of the whole operation by considering the preconditions of its constituent parts.

Appendix D provides a commentary on the tracing of this document to the SRS [2]. It also lists those requirements from the SRS that do not trace to the body of this specification. These are categorised by the reason for exclusion.

Appendix E provides a commentary on the tracing of this document to the Security Target [4].

2.2 Trace units

Each section of the specification has been tagged with a named *traceunit* which will be used as a reference from later design documents. All trace units in this document have the prefix “FS” identifying them as originating in the Formal Specification.

Most traceunits contain a list of requirements that are relevent to that part of the specification. These are taken from the SRS [2] and Security Target [4].

For example consider the traceunit on page 12. Here the section on tokens is identified by the name *FS.Types.Tokens* and this section is relevant to the satisfaction of a number of requirements from the Protection Profile [3] including *FCO_NRO.2.1*.

2.3 Z basics

This formal specification is written using the Z formal notation. [1]

2.3.1 Z comments

The intention is that someone unfamiliar with Z should be able to read this specification and gain a complete understanding of the functionality of the TIS system specified within.

We have attempted to make the informal commentary as complete and unambiuous as possible. We have also separated out the parts of the commentary that are only relevant for understanding the formal model, as below:

- Readers who are not interested in the formal model can skip these sections of the commentary.

2.3.2 Reading Z

Readers of this specification are encouraged to read the Z formal notation. Reading the Z in the context of the commentary should disambiguate the English.

In Appendix A we explain the basics of how to read Z. These basic ideas should be sufficient to aid reading this specification. For a more detailed description of the Z notation refer to [1].

2.3.3 Defining Optional Items

In order to be able to define optional items we make the following definitions.

$$\mathrm{optional}\, X == \{x : \mathbb{F}\, X \mid \#x \leq 1\}$$
$$nil[X] == \varnothing[X]$$
$$the\,[X] == \{\, x : X \bullet \{x\} \mapsto x\}$$

2.4 TIS Basic Types

FS.Types.Time

Time and date is some universal clock, which for our purposes can be modelled as just the naturals.

$$TIME == \mathbb{N}$$

We define a constant *zeroTime* used at system initialisation.

$$zeroTime == 0$$

FS.Types.Presence

Many entities such as tokens, fingers and floppy disks may be presented to the system and removed by the user. We monitor the presence of these entities.

$$PRESENCE ::= present \mid absent$$

FS.Types.Clearance

CLASS is the ordered classifications on document, areas, and people.

$$CLASS ::= unmarked \mid unclassified \mid restricted \mid confidential \mid secret \mid topsecret$$

There may be other aspects to classification but these are not modelled here.

Clearance

$$class : CLASS$$

▷ See: *CLASS* (p. 8)

There is an ordering on the type *Clearance*. The function *minClearance* gives the minimum of a pair of elements of type *Clearance*. This will be the Clearance with the lowest class. The ordering on class is formally defined within the design, informally *unmarked* is the lowest class and *topsecret* is the highest class.

$$minClearance : Clearance \times Clearance \longrightarrow Clearance$$

▷ See: *Clearance* (p. 8)

FS.Types.Privilege

PRIVILEGE is the role held by the Token user. This will determine the privileges that the Token user has when interacting with the ID station.

$$PRIVILEGE ::= userOnly \mid guard \mid securityOfficer \mid auditManager$$

FS.Types.User

A *USER* is a unique identification of a certificate owner. For the purpose of this specification it is a given type.

$$[USER]$$

FS.Types.Issuer

An *ISSUER* is a unique identification of an issuing body. Issuers are privileged users with the ability to issue certificates.

$$ISSUER : \mathbb{P}\, USER$$

FS.Types.Fingerprint

FINGERPRINT will need to include sufficient control information to allow us to compare with templates and decide a match or not.

$$[FINGERPRINT]$$

FS.Types.FingerprintTemplate

A *FINGERPRINTTEMPLATE* contains abstracted information, derived from a number of sample readings of a fingerprint.

$$[FINGERPRINTTEMPLATE]$$

The fingerprint template and will be accompanied by additional information, such as the threshold level to be applied to any comparisons. This is not currently modelled.

FingerprintTemplate

$$template : FINGERPRINTTEMPLATE$$

2.5 Keys and Encryption

FS.KeyTypes.Keys

The signing and validation of certificates used in Tokeneer relies on the use of asymmetric keys, which comprise two parts, one which is public and one which is private.

$[KEYPART]$

Certificates are signed by an issuer using the private part, and can be verified by anyone who holds the public part.

Abstractly, only the public part is visible, and it is the only part we need to model. In the design we will introduce the private part too.

Knowing an issuer is equivalent to having a copy of the issuer's public key part. While possessing an issuer's private key part means that you are that issuer.

2.6 Certificates, Tokens and Enrolment Data

2.6.1 Certificates

FS.Types.Certificates

All certificates consist of data and a signature. A number of attributes are encoded within the data. Some attributes are common to all certificates.

All certificates can be uniquely identified by their issuer and the serial number supplied by the issuer when the certificate is created. The only aspect of the certificate ID which is significant at this level is the issuer, so we will model the certificate ID as containing an *ISSUER* only.

CertificateId

$issuer : ISSUER$

▷ See: *ISSUER* (p. 9)

In addition to the unique certificate id all certificates contain a validity period during which time they are valid. We will model this validity period as a set of *TIME*s during which they are valid, which is more general and easier to state.

Each certificate is signed and can be verified using a key, typically the public key of an issuer. We model this by associating with each certificate the key required to validate the certificate. Note that the key is optional since in the case that the signature or data is corrupt, no key will validate the certificate.

Certificate

$id : CertificateId$

$validityPeriod : \mathbb{P}\ TIME$

$isValidatedBy : \mathrm{optional}\ KEYPART$

▷ See: *CertificateId* (p. 10), *TIME* (p. 8), optional (p. 8)

Each type of certificate potentially expands on these attributes.

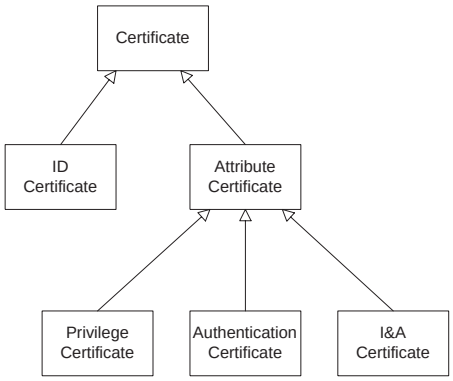

Figure 2.1: Hierarchy of certificate types

The ID certificate is an X.509 certificate. ID certificates are used during enrolment as well as being present on tokens.

The subject is the name of the entity being identified by the certificate and the key is the entity's public key.

We don't need to know about the key of the Token unless we implement the TOKENEER Authentication Protocol or some other secure comminications protocol between TIS and the Token. Secure communications with the Token are outside the current scope of this system.

IDCert

$$\begin{array}{l} \mathit{Certificate} \\ \mathit{subject} : \mathit{USER} \\ \mathit{subjectPubK} : \mathit{KEYPART} \end{array}$$

▷ See: *Certificate* (p. 10)

In general an ID certificate is not validated by the keypart held on the certificate.

The ID Certificate of a CA (Certification Authority) is a root certificate and is signed by itself. The chain of trust has to start somewhere.

CAIdCert

$$\begin{array}{l} \mathit{IDCert} \\ \hline \mathit{isValidatedBy} = \{\, \mathit{subjectPubK} \,\} \end{array}$$

▷ See: *IDCert* (p. 11)

The certificates containing attributes all share some common attributes.

All attribute certificates contain the ID of the Token and the identification of the ID certificate. Specific types of attribute certificate build on this common structure.

AttCertificate

Certificate
baseCertId : *CertificateId*
tokenID : *TOKENID*

▷ See: *Certificate* (p. 10), *CertificateId* (p. 10)

A privilege certificate additionally contains a role and clearance.

PrivCert

AttCertificate
role : *PRIVILEGE*
clearance : *Clearance*

▷ See: *AttCertificate* (p. 12), *PRIVILEGE* (p. 9), *Clearance* (p. 8)

An authorisation certificate has the same structure as a privilege certificate.

AuthCert

AttCertificate
role : *PRIVILEGE*
clearance : *Clearance*

▷ See: *AttCertificate* (p. 12), *PRIVILEGE* (p. 9), *Clearance* (p. 8)

An I&A (identification and authentication) certificate additionally contains a fingerprint template.

IandACert

AttCertificate
template : *FingerprintTemplate*

▷ See: *AttCertificate* (p. 12), *FingerprintTemplate* (p. 9)

2.6.2 Tokens

FS.Types.Tokens

FCO_NRO.2.1	*FDP_DAU.2.2*
FCO_NRO.2.2	*FIA_UAU.3.1*
FCO_NRO.2.3	*FAI_UAU.3.2*
FDP_DAU.2.1	

▷ Refer to Section 2.2 for explanation of the above tracing block.

Each Token has a unique ID, ensured unique by the smartcard supplier.

[*TOKENID*]

A *Token* contains a number of certificates. The authorisation certificate is optional while the others must be present.

Token

$$
\begin{array}{l}
tokenID : TOKENID \\
\\
idCert : IDCert \\
privCert : PrivCert \\
iandACert : IandACert \\
authCert : \mathrm{optional}\, AuthCert
\end{array}
$$

▷ See: *IDCert* (p. 11), *PrivCert* (p. 12), *IandACert* (p. 12), optional (p. 8), *AuthCert* (p. 12)

A *Token* is valid if all of the certificates on it are well-formed, each certificate correctly cross-references to the ID Certificate, and each certificate correctly cross-references to the *Token* ID.

A token need not contain a valid Authorisation certificate to be considered valid.

ValidToken

$$
\begin{array}{l}
Token \\
\hline
privCert.baseCertId = idCert.id \\
iandACert.baseCertId = idCert.id \\
\\
privCert.tokenID = tokenID \\
iandACert.tokenID = tokenID
\end{array}
$$

▷ See: *Token* (p. 13)

If the Authorisation certificate is present it will only be used if it is valid, in that it correctly cross-references to the *Token* ID and the ID Certificate.

TokenWithValidAuth

$$
\begin{array}{l}
Token \\
\hline
authCert \neq nil \\
\quad \land\ (the\ authCert).tokenID = tokenID \\
\quad \land\ (the\ authCert).baseCertId = idCert.id
\end{array}
$$

▷ See: *Token* (p. 13), *nil* (p. 8), *the* (p. 8)

A *Token* is current if all of the Certificates are current, or if only the Auth Cert is non-current. Currency needs a time, which is included in the schema, and will need to be tied to the relevent time when this schema is used.

CurrentToken

$$
\begin{array}{l}
ValidToken \\
now : TIME \\
\hline
now \in idCert.validityPeriod \\
\quad \cap\ privCert.validityPeriod \\
\quad \cap\ iandACert.validityPeriod
\end{array}
$$

▷ See: *ValidToken* (p. 13), *TIME* (p. 8)

2.6.3 Enrolment Data

FS.Types.Enrolment	
FMT_MSA.2.1	*FMT_MTD.3.1*

Enrolment data is the information the ID station needs in order to know how to authenticate tokens presented to it, and to produce its own authentication certificates such that they can be authenticated by workstations in the enclave.

Enrolment data consists of a number of ID certificates:

- this ID Station's ID Certificate, which will be signed by a CA.
- A number of other Issuers' ID Certificates. These will belong to
  - CAs, who authenticate AAs (Attribute Authorities) and ID Stations. These will be self signed.
  - AAs, who authenticate privilege and I&A certificates.

The ID Station's certificate is just one of the issuer certificates, although we will want to be able to identify it as belonging to this ID station.

Enrol

$$idStationCert : IDCert$$
$$issuerCerts : \mathbb{P}\, IDCert$$

$$idStationCert \in issuerCerts$$

▷ See: *IDCert* (p. 11)

For the Enrolment data to be considered valid each certificate must be signed correctly and the Issuer's certificate must be present for it to be possible to check that the signatures are correct. Note that CA ID certificates are self signed but AA and IDStation certificates are signed by an CA.

ValidEnrol

Enrol

$$issuerCerts \cap \{CAIdCert\} \neq \varnothing$$

$$\forall\, cert : issuerCerts \bullet$$
$$\quad cert.isValidatedBy \neq nil$$
$$\quad \wedge\ (\exists\, issuerCert : issuerCerts \bullet issuerCert \in CAIdCert$$
$$\qquad \wedge\ the\, cert.isValidatedBy = issuerCert.subjectPubK$$
$$\qquad \wedge\ cert.id.issuer = issuerCert.subject)$$

▷ See: *Enrol* (p. 14), *CAIdCert* (p. 11), *nil* (p. 8), *the* (p. 8)

▷ There must be an ID certificate for at least one CA.

▷ For each certificate the enrolment data must include the ID certificate for the issuer of the certificate, the certificate must be validated by the issuer's key and the issuer of the certificate must be a CA.

2.7 World outside the ID Station

We choose to model the real world (or at least the real peripherals) as being outside the ID Station. When the user inserts a token, they are providing input to the ID Station. It is up to the ID Station to then respond by reading the real world input into its own, internal representation. The ID Station receives stimulus from the real world and itself changes the real world. All real world entities are modelled as components of the *RealWorld*.

We will distingush between real world entities that we use (eg. *finger*), we control (eg. *alarm*) and we may change (eg. *userToken* or *floppy*).

2.7.1 Real World types

FS.Types.RealWorld

There are several types associated with the real world. The door, latch and alarm all have two possible states.

$DOOR ::= open \mid closed$
$LATCH ::= unlocked \mid locked$
$ALARM ::= silent \mid alarming$

Display messages are the short messages presented to the user on the small display outside the enclave.

$DISPLAYMESSAGE ::= blank \mid welcome \mid insertFinger \mid openDoor \mid wait \mid removeToken \mid tokenUpdateFailed \mid doorUnlocked$

The messages that appear on the display are presented in table 2.1.

	Displayed text	
Message	**Top line**	**Bottom line**
blank	SYSTEM NOT OPERATIONAL	
welcome	WELCOME TO TIS	ENTER TOKEN
insertFinger	AUTHENTICATING USER	INSERT FINGER
wait	AUTHENTICATING USER	PLEASE WAIT
openDoor		REMOVE TOKEN AND ENTER
removeToken	ENTRY DENIED	REMOVE TOKEN
tokenUpdateFailed		TOKEN UPDATE FAILED
doorUnlocked		ENTER ENCLAVE

Table 2.1: Display Messages

Because it is possible to be trying to read a token that is not inserted, or a fingerprint when no finger is inserted, or an invalid token or fingerprint, we introduce free types to capture the absence or poor quality of these.

The values *badFP* and *badT* represent all possible error codes that occur when trying to capture this data. The system will behave the same way in all failure cases with only the audit log capturing the different error codes that actually occur.

$FINGERPRINTTRY ::= noFP \mid badFP \mid goodFP\langle\langle FINGERPRINT\rangle\rangle$
$TOKENTRY ::= noT \mid badT \mid goodT\langle\langle Token\rangle\rangle$

▷ See: *Token* (p. 13)

When modelling data supplied on a floppy disk we model the posibility of the disk not being present, being empty or being corrupt as well as containing valid data. We make the assumption that each floppy disk will only contain one data file, either enrolment data, configuration data or audit data.

$$FLOPPY ::= noFloppy \mid emptyFloppy \mid badFloppy \mid enrolmentFile\langle\langle ValidEnrol\rangle\rangle \mid auditFile\langle\langle \mathbb{F}\, Audit\rangle\rangle \mid configFile\langle\langle Config\rangle\rangle$$

▷ See: *ValidEnrol* (p. 14)

Inputs may be supplied by an administrator at the keyboard. We model input values representing no data, invalid data or a valid request to perform an adminstrator operation.

$$KEYBOARD ::= noKB \mid badKB \mid keyedOps\langle\langle ADMINOP\rangle\rangle$$

There are a number of messages that may appear on the TIS screen within the enclave. Some of these are simple messages, the text of these is supplied in Table 2.2. Others involve more complex presentation of data, such as configuration data or system statistics, the details of this presentation is left to design.

$$SCREENTEXT ::= clear \mid welcomeAdmin \mid busy \mid removeAdminToken \mid closeDoor \mid requestAdminOp \mid doingOp \mid invalidRequest \mid invalidData \mid insertEnrolmentData \mid validatingEnrolmentData \mid enrolmentFailed \mid archiveFailed \mid insertBlankFloppy \mid insertConfigData \mid displayStats\langle\langle Stats\rangle\rangle \mid displayConfigData\langle\langle Config\rangle\rangle$$

In addition to the messages statistics and the current configuration data may be displayed on the screen.

Screen

$$screenStats : SCREENTEXT$$
$$screenMsg : SCREENTEXT$$
$$screenConfig : SCREENTEXT$$

▷ See: *SCREENTEXT* (p. 16)

2.7.2 The Real World

Within this section we consider the entities with which TIS will interact at an abstract level. We do not consider protocol information or any flows of information that are not visible to an external observer. For instance typical fingerprint readers need to have stale data cleared by TIS to ensure that TIS always reads fresh data. This is not modelled in this specification but will be introduced during the design.

The real world entities that are controlled by TIS are as follows:

- the latch on the door into the enclave.

Message	**Displayed text**
clear	
welcomeAdmin	WELCOME TO TIS
busy	SYSTEM BUSY PLEASE WAIT
removeAdminToken	REMOVE TOKEN
closeDoor	CLOSE ENCLAVE DOOR
requestAdminOp	ENTER REQUIRED OPERATION
doingOp	PERFORMING OPERATION PLEASE WAIT
invalidRequest	INVALID REQUEST: PLEASE ENTER NEW OPERATION
invalidData	INVALID DATA: PLEASE ENTER NEW OPERATION
archiveFailed	ARCHIVE FAILED: PLEASE ENTER NEW OPERATION
insertEnrolmentData	PLEASE INSERT ENROLMENT DATA FLOPPY
validatingEnrolmentData	VALIDATING ENROLMENT DATA PLEASE WAIT
enrolmentFailed	INVALID ENROLMENT DATA
insertBlankFloppy	INSERT BLANK FLOPPY
insertConfigData	INSERT CONFIGURATION DATA FLOPPY

Table 2.2: Short Screen Messages

- the audible alarm.
- the display that resides outside the enclave.
- the screen on the ID Station within the enclave with which the administrator interacts.

TISControlledRealWorld

latch : *LATCH*
alarm : *ALARM*
display : *DISPLAYMESSAGE*
screen : *Screen*

▷ See: *LATCH* (p. 15), *ALARM* (p. 15), *DISPLAYMESSAGE* (p. 15), *Screen* (p. 16)

The real world entities that are used by TIS are as follows:

- the real world has a concept of time. This is taken from an external time source.
- the door into the enclave that is monitored by the ID Station.
- fingerprints are read, via the biometric reader, into the ID Station for comparison with fingerprint templates.
- a user, trying to enter the enclave will supply their token to the ID station via the token reader that resides outside the enclave.
- a user within the enclave who has administrator privileges will supply their token to the ID station via the token reader that resides inside the enclave.
- the ID Station accepts enrolment data and configuration data on a floppy disk. The disk drive resides in the enclave.
- the ID Station has a keyboard within the enclave which the administrator uses to control TIS.

TISMonitoredRealWorld

$now : TIME$
$door : DOOR$
$finger : FINGERPRINTTRY$
$userToken, adminToken : TOKENTRY$
$floppy : FLOPPY$
$keyboard : KEYBOARD$

▷ See: *TIME* (p. 8), *DOOR* (p. 15), *FINGERPRINTTRY* (p. 15), *TOKENTRY* (p. 15), *FLOPPY* (p. 16), *KEYBOARD* (p. 16)

In addition TIS may change some of the entities that it uses from the real world.

- The ID station may need to update the *userToken* token (with an Authentication Certificate).
- the ID Station archives the Audit Log to floppy disk so may write to *floppy*.

The Whole real world is given by:

$$RealWorld \mathrel{\widehat=} TISControlledRealWorld \land TISMonitoredRealWorld$$

▷ See: *TISControlledRealWorld* (p. 17), *TISMonitoredRealWorld* (p. 18)

3 THE TOKEN ID STATION

TIS maintains various state components, these are described and elaborated within this section.

3.1 Configuration Data

FS.ConfigData.State

Config will be a structure with all the configuration data. Configuration data can only be modified by an administrator. This data includes:

- Durations for internal timeouts. These effect how long the system waits before raising an audible alarm, how long the system leaves the door unlocked for, and how long the system waits for a successful token removal.
- The security classification of the enclave.
- The rules for allocating validity periods to authorisation certificates. These rules will depend on the time at which the certificate was issued, and may also depend on the role of the user, for example some roles may not be given use of the workstations “out of hours”.
- The rules for allowing entry to the enclave. These rules will depend on the role and security classification of the user, for example some roles may not be given access to the enclave “out of hours”.
- The minimum size of the audit log before truncation may occur, *minPreservedLogSize*, which is configured to be within available file store capacity of the TIS. A slightly smaller value, *alarmThresholdSize*, sets the size of the audit log at which an alarm is raised, with the intention that the audit log will be archived and cleared before the truncation occurs. We acknowledge that there will be a system limit which affects the largest size of log that can be guaranteed to be preserved.

$$maxSupportedLogSize : \mathbb{N}$$

Config

$$\begin{array}{l} alarmSilentDuration, latchUnlockDuration : TIME \\ tokenRemovalDuration : TIME \\ enclaveClearance : Clearance \\ authPeriod : PRIVILEGE \longrightarrow TIME \longrightarrow \mathbb{P}\ TIME \\ entryPeriod : PRIVILEGE \longrightarrow CLASS \longrightarrow \mathbb{P}\ TIME \\ minPreservedLogSize : \mathbb{N} \\ alarmThresholdSize : \mathbb{N} \\ \hline alarmThresholdSize < minPreservedLogSize \\ minPreservedLogSize \leq maxSupportedLogSize \end{array}$$

▷ See: *TIME* (p. 8), *Clearance* (p. 8), *PRIVILEGE* (p. 9), *CLASS* (p. 8), *maxSupportedLogSize* (p. 19)

In practice there will be constraints on the authorisation periods and entry periods. These constraints will be considered during the design. There will also be constraints on the maximum FAR permitted by the biometic verification. This will be introduced in the design.

3.2 Audit Log

FS.AuditLog.State

TIS maintains an audit log. This is a log of all auditable events and actions performed or monitored by TIS. The audit log will be used to analyse the interactions with the TIS.

Audit will be a structure for each audit record, recording at least time of event, type of event, user if known. We use title case because we know this is a type we will be elaborating later.

$$[Audit]$$

Each audit element has associated with it a size, which may vary between audit elements. The size of an audit log can be determined from the size of its elements.

$$\begin{array}{l} sizeElement : Audit \longrightarrow \mathbb{N} \\ sizeLog : \mathbb{F}\, Audit \longrightarrow \mathbb{N} \\ \hline sizeLog\varnothing = 0 \\ \forall\, log : \mathbb{F}\, Audit; entry : Audit \bullet \\ \quad entry \in log \Rightarrow sizeLog\, log = sizeLog\, (log \setminus \{entry\}) + sizeElement\, entry \end{array}$$

The Audit log consists of a number of *Audit* elements. An audit error alarm will be raised if the audit log becomes full and needs to be archived and cleared.

$$\begin{array}{l} AuditLog \\ \hline auditLog : \mathbb{F}\, Audit \\ auditAlarm : ALARM \end{array}$$

▷ See: *ALARM* (p. 15)

All audit elements have associated with them a timestamp so it is possible to determine the times of the newest and oldest entries in the log.

$$\begin{array}{l} oldestLogTime : \mathbb{F}\, Audit \longrightarrow TIME \\ newestLogTime : \mathbb{F}\, Audit \longrightarrow TIME \\ \hline \forall\, A, B : \mathbb{F}\, Audit \bullet \\ \quad newestLogTime(A \cup B) \geq newestLogTime\, A \\ \quad \land\ oldestLogTime(A \cup B) \leq oldestLogTime\, A \end{array}$$

▷ See: *TIME* (p. 8)

▷ Both these functions are monotonic. In particular the *newestLogTime* $\varnothing$ is the earliest time and the *oldestLogTime* $\varnothing$ is the latest possible time.

3.3 Key Store

FS.KeyStore.State

TIS maintains a key store which contains all Issuer keys relevant to its function. This will include known CAs, AAs and its own key. Once enrolled, the key store also contains the ID station's name and own key.

KeyStore

$$issuerKey : ISSUER \pfun KEYPART$$
$$ownName : \mathrm{optional}\, ISSUER$$

$$ownName \neq nil \Rightarrow the\, ownName \in \mathrm{dom}\, issuerKey$$

▷ See: *ISSUER* (p. 9), optional (p. 8), *nil* (p. 8), *the* (p. 8)

▷ An ID Station is issued with a name at enrolment. Prior to enrolment it will not have a name.

▷ This ID Station, once named, will have its key held with the other issuers' keys.

3.4 System Statistics

FS.Stats.State

TIS keeps track of the number of times that a entry to the enclave has been attempted (and denied) and the number of times it has succeeded. It also records the number of times that a biometric comparison has been made (and failed) and the number of times it succeeded.

By retaining these statistics it is possible for the performance of the system to be monitored.

Stats

$$successEntry : \mathbb{N}$$
$$failEntry : \mathbb{N}$$
$$successBio : \mathbb{N}$$
$$failBio : \mathbb{N}$$

3.5 Administration

FS.Admin.State

SFP_DAC	*FMT_MOF.1.1*
FDP_ACC.1.1	*FIA_USB.1.1*
FDP_ACF.1.1	*FMT_MSA.1.1*
FDP_ACF.1.2	*FMT_MTD.1.1*
FDP_ACF.1.3	*FMT_SMR.2.1*
FDP_ACF.1.4	*FMT_SAE.1.1*

In addition to its role of authorising entry to the enclave, TIS supports a number of administrative operations.

- ArchiveLog - writes the archive log to floppy and truncates the internally held archive log.
- UpdateConfiguration - accepts new configuration data from a floppy.
- OverrideDoorLock - unlocks the enclave door.
- Shutdown - stops TIS, leaving the protected entry to the enclave secure.

$ADMINOP ::= archiveLog \mid updateConfigData \mid overrideLock \mid shutdownOp$

Other operations that could be supported are *DisplayLog*, *CancelAlarm*, *ClearStats*, *Decommission*, *AddIssuers*, *RemoveIssuers*. These additional operations will be considered out of scope of this re-development.

Only users with administrator privileges can make use of the TIS to perform administrative functions. There are a number of different administrator privileges that may be held.

$ADMINPRIVILEGE == \{guard, auditManager, securityOfficer\}$

▷ See: *guard* (p. 9), *auditManager* (p. 9), *securityOfficer* (p. 9)

The role held by the administrator will determine the operations available to the administrator. An administrator can only hold one role.

Admin

$rolePresent : \mathrm{optional}\, ADMINPRIVILEGE$
$availableOps : \mathbb{P}\, ADMINOP$
$currentAdminOp : \mathrm{optional}\, ADMINOP$

$rolePresent = nil \Rightarrow availableOps = \varnothing$
$(rolePresent \neq nil \land the\, rolePresent = guard) \Rightarrow availableOps = \{overrideLock\}$
$(rolePresent \neq nil \land the\, rolePresent = auditManager) \Rightarrow availableOps = \{archiveLog\}$
$(rolePresent \neq nil \land the\, rolePresent = securityOfficer) \Rightarrow availableOps = \{updateConfigData, shutdownOp\}$
$currentAdminOp \neq nil \Rightarrow$
$\quad (the\, currentAdminOp \in availableOps \land rolePresent \neq nil)$

▷ See: optional (p. 8), *ADMINPRIVILEGE* (p. 22), *ADMINOP* (p. 22), *nil* (p. 8), *the* (p. 8), *guard* (p. 9), *overrideLock* (p. 22), *auditManager* (p. 9), *archiveLog* (p. 22), *securityOfficer* (p. 9), *updateConfigData* (p. 22), *shutdownOp* (p. 22)

▷ The *availableOps* are completely determined by the roles present.

In order to perform an administrative operation an administrator must be present. Presence will be determined by an appropriate token being present in the administrator's card reader.

3.6 Real World Entities

FS.RealWorld.State	
FAU_ARP.1.1	*FAU_SAA.1.2*
FAU_SAA.1.1	*FPT_RVM.1.1*

The latch is allowed to be in two states: *locked* and *unlocked*. When the latch is unlocked, *latchTimeout* will be set to the time at which the lock must again be *locked*.

The alarm is similar to the latch, in that it has a *silent*, and *alarming*, with an *alarmTimeout*. Once the door and latch move into a potentially insecure state (door *open* and latch *locked*) then the *alarmTimeout* is set to the time at which the alarm will sound.

The state of *currentLatch* is entirely derived from whether the *latchTimeout* has fired or not. The state of *doorAlarm* is also entirely derived — if the state is potentially insecure and the *alarmTimeout* has fired, the alarm must be *alarming*.

DoorLatchAlarm

$$
\begin{array}{l}
currentTime : TIME \\
currentDoor : DOOR \\
currentLatch : LATCH \\
doorAlarm : ALARM \\
latchTimeout : TIME \\
alarmTimeout : TIME \\
\hline
currentLatch = locked \Leftrightarrow currentTime \geq latchTimeout \\
doorAlarm = alarming \Leftrightarrow \\
\quad (currentDoor = open \\
\qquad \wedge\ currentLatch = locked \\
\qquad \wedge\ currentTime \geq alarmTimeout)
\end{array}
$$

▷ See: *TIME* (p. 8), *DOOR* (p. 15), *LATCH* (p. 15), *ALARM* (p. 15), *locked* (p. 15), *alarming* (p. 15), *open* (p. 15)

The ID Station holds internal representations of all of the Real World, plus its own data. It holds separate indications of the presence of input in the real world peripherals of the User Token, Admin Token, Fingerprint reader, and Floppy disk. This is so that once the input has been read, and the card, finger or disk removed, the ID Station can continue to know what the value was, even if it later detects that the real world entity has been removed.

UserToken

$$
\begin{array}{l}
currentUserToken : TOKENTRY \\
userTokenPresence : PRESENCE
\end{array}
$$

▷ See: *TOKENTRY* (p. 15), *PRESENCE* (p. 8)

AdminToken

$$
\begin{array}{l}
currentAdminToken : TOKENTRY \\
adminTokenPresence : PRESENCE
\end{array}
$$

▷ See: *TOKENTRY* (p. 15), *PRESENCE* (p. 8)

Finger

$$
\begin{array}{l}
currentFinger : FINGERPRINTTRY \\
fingerPresence : PRESENCE
\end{array}
$$

▷ See: *FINGERPRINTTRY* (p. 15), *PRESENCE* (p. 8)

We need to retain an internal view of the last data written to the floppy as well as the current data on the floppy, this is because we need to check that writing to floppy works when we archive the log.

Floppy

$$
\begin{array}{l}
currentFloppy : FLOPPY \\
writtenFloppy : FLOPPY \\
floppyPresence : PRESENCE
\end{array}
$$

▷ See: *FLOPPY* (p. 16), *PRESENCE* (p. 8)

Keyboard

currentKeyedData : *KEYBOARD*
keyedDataPresence : *PRESENCE*

▷ See: *KEYBOARD* (p. 16), *PRESENCE* (p. 8)

3.7 Internal State

FS.Internal.State
FPT_RVM.1.1

STATUS and *ENCLAVESTATUS* are purely internal records of the progress through processing. *STATUS* tracks progress through user entry, while *ENCLAVESTATUS* tracks progress through all activities performed within the enclave.

STATUS ::= *quiescent* |

gotUserToken | *waitingFinger* | *gotFinger* | *waitingUpdateToken* | *waitingEntry* | *waitingRemoveTokenSuccess* | *waitingRemoveTokenFail*

ENCLAVESTATUS ::= *notEnrolled* | *waitingEnrol* | *waitingEndEnrol* |

enclaveQuiescent |

gotAdminToken | *waitingRemoveAdminTokenFail* | *waitingStartAdminOp* | *waitingFinishAdminOp* |

shutdown

The states *quiescent* and *enclaveQuiescent* represent the enclave interface and the user entry interface being quiescent.

The states *gotUserToken, .. waitingRemoveTokenFail* are all associated with the process of user authentication and entry. These are described futher in Section 6.

The states *notEnrolled, .. waitingEnrolEnd* reflect enrolment activity that must be performed before TIS can offer any of its normal operations. Once the TIS is successfully enrolled it becomes *quiescent*.

The states *gotAdminToken, .. waitingFinishAdminOp* reflect activity at the TIS console relating to administrator use of TIS.

The state *shutdown* models the system when it is shutdown.

Internally the system maintains knowledge of the status of the user entry operation and the enclave. It also holds a timeout which is only relevant when the status is on *waitingRemoveTokenSuccess*.

Internal

status : *STATUS*
enclaveStatus : *ENCLAVESTATUS*
tokenRemovalTimeout : *TIME*

▷ See: *STATUS* (p. 24), *ENCLAVESTATUS* (p. 24), *TIME* (p. 8)

3.8 The whole Token ID Station

FS.TIS.State

The whole Token ID Station is constructed from combining the described state components.

In addition there is a display outside the enclave and screen within the enclave. The ID Station screen within the enclave may display many pieces of information. The majority of this data will be determined by state invariants.

The alarm, door and latch conform to their consistency rules. The relationships between available operations and roles present are preserved.

If the authentication protocol has moved on to requesting a fingerprint, then the User Token will have passed its validation checks.

Similarly if the system considers there to be an administrator present then the Admin Token will have passed its validation checks.

Once the ID station has been enrolled it has a name.

TIS is only ever in the two states *waitingStartAdminOp* or *waitingFinishAdminOp* when then there is a current admin operation in progress. For single phase operations the state *waitingFinishAdminOp* is not used.

TIS will only read the Admin Token to log on an administrator if there is not an administrator role currently present.

IDStation

$$
\begin{array}{l}
UserToken \\
AdminToken \\
Finger \\
DoorLatchAlarm \\
Floppy \\
Keyboard \\
Config \\
Stats \\
KeyStore \\
Admin \\
AuditLog \\
Internal \\
\\
currentDisplay : DISPLAYMESSAGE \\
currentScreen : Screen \\
\hline
status \in \{ gotFinger, waitingFinger, waitingUpdateToken, waitingEntry \} \Rightarrow \\
\quad ((\exists\, ValidToken \bullet goodT(\theta ValidToken) = currentUserToken) \\
\qquad \vee\ (\exists\, TokenWithValidAuth \bullet goodT(\theta TokenWithValidAuth) = currentUserToken)) \\
\\
rolePresent \neq nil \Rightarrow \\
\quad (\exists\, TokenWithValidAuth \bullet goodT(\theta TokenWithValidAuth) = currentAdminToken) \\
\\
enclaveStatus \notin \{ notEnrolled, waitingEnrol, waitingEndEnrol \} \Rightarrow \\
\quad (ownName \neq nil) \\
\\
enclaveStatus \in \{ waitingStartAdminOp, waitingFinishAdminOp \} \Leftrightarrow currentAdminOp \neq nil \\
\\
(currentAdminOp \neq nil \wedge the\, currentAdminOp \in \{ shutdownOp, overrideLock \}) \\
\qquad \Rightarrow enclaveStatus = waitingStartAdminOp \\
\\
enclaveStatus = gotAdminToken \Rightarrow rolePresent = nil \\
\\
currentScreen.screenStats = displayStats(\theta Stats) \\
currentScreen.screenConfig = displayConfigData(\theta Config)
\end{array}
$$

▷ See: *UserToken* (p. 23), *AdminToken* (p. 23), *Finger* (p. 23), *DoorLatchAlarm* (p. 22), *Floppy* (p. 23), *Keyboard* (p. 24), *Config* (p. 19), *Stats* (p. 21), *KeyStore* (p. 21), *Admin* (p. 22), *AuditLog* (p. 20), *Internal* (p. 24), *DISPLAYMESSAGE* (p. 15), *Screen* (p. 16), *gotFinger* (p. 24), *waitingFinger* (p. 24), *waitingUpdateToken* (p. 24), *waitingEntry* (p. 24), *ValidToken* (p. 13), *goodT* (p. 15), *TokenWithValidAuth* (p. 13), *nil* (p. 8), *notEnrolled* (p. 24), *waitingEnrol* (p. 24), *waitingEndEnrol* (p. 24), *waitingStartAdminOp* (p. 24), *waitingFinishAdminOp* (p. 24), *the* (p. 8), *shutdownOp* (p. 22), *overrideLock* (p. 22), *displayStats* (p. 16), *displayConfigData* (p. 16)

▷ Note that the token may not still be current since time will have moved on since the checks were performed.

▷ Operations that can be performed in a single phase do not result in TIS entering the state *waitingFinishAdminOp* as they are finished when they are started.

▷ TIS only enters the state *gotAdminToken* when there is no administrator present.

▷ Invariants define many of the screen components.

4 OPERATIONS INTERFACING TO THE ID STATION

4.1 Real World Changes

The monitored components of the real world can change at any time. The only assumption we make of the real world is that the time supplied by the external time source increases. If the external time source does not supply increasing times then our system is not guaranteed to work.

RealWorldChanges

$$\Delta RealWorld$$

$$now' \geq now$$

▷ See: *RealWorld* (p. 18)

4.2 Obtaining inputs from the real world

In this model all data is polled from the real world on a periodic basis.

4.2.1 Polling the real world

FS.Interface.TISPoll
FPT_STM.1.1

We poll all of the real world entities.

Changes to the time, may affect the state of the latch.

PollTime

$$\Delta DoorLatchAlarm$$
$$RealWorld$$

$$currentTime' = now$$

▷ See: *DoorLatchAlarm* (p. 22), *RealWorld* (p. 18)

When polling the door, we do not change the alarm timeout or latch timeout. The internal representation of the latch or the alarm may change as a result of changes to the attributes that influence their values.

PollDoor

$$\Delta DoorLatchAlarm$$
$$RealWorld$$

$$currentDoor' = door$$
$$latchTimeout' = latchTimeout$$
$$alarmTimeout' = alarmTimeout$$

▷ See: *DoorLatchAlarm* (p. 22), *RealWorld* (p. 18)

The system polls the tokens, finger, floppy and keyboard and the last present value is stored. This allows the peripheral to be removed before TIS has completed use of the data.

PollUserToken

$\Delta UserToken$
RealWorld

$userTokenPresence' = present \Leftrightarrow userToken \neq noT$
$currentUserToken' = \mathbf{if}\ userToken \neq noT\ \mathbf{then}\ userToken\ \mathbf{else}\ currentUserToken$

▷ See: *UserToken* (p. 23), *RealWorld* (p. 18), *present* (p. 8), *noT* (p. 15)

PollAdminToken

$\Delta AdminToken$
RealWorld

$adminTokenPresence' = present \Leftrightarrow adminToken \neq noT$
$currentAdminToken' = \mathbf{if}\ adminToken \neq noT\ \mathbf{then}\ adminToken\ \mathbf{else}\ currentAdminToken$

▷ See: *AdminToken* (p. 23), *RealWorld* (p. 18), *present* (p. 8), *noT* (p. 15)

PollFinger

$\Delta Finger$
RealWorld

$fingerPresence' = present \Leftrightarrow finger \neq noFP$
$currentFinger' = \mathbf{if}\ finger \neq noFP\ \mathbf{then}\ finger\ \mathbf{else}\ currentFinger$

▷ See: *Finger* (p. 23), *RealWorld* (p. 18), *present* (p. 8), *noFP* (p. 15)

PollFloppy

$\Delta Floppy$
RealWorld

$floppyPresence' = present \Leftrightarrow floppy \neq noFloppy$
$currentFloppy' = \mathbf{if}\ floppy \neq noFloppy\ \mathbf{then}\ floppy\ \mathbf{else}\ currentFloppy$
$writtenFloppy' = writtenFloppy$

▷ See: *Floppy* (p. 23), *RealWorld* (p. 18), *present* (p. 8), *noFloppy* (p. 16)

PollKeyboard

$\Delta Keyboard$
RealWorld

$keyedDataPresence = present \Leftrightarrow keyboard \neq noKB$
$currentKeyedData' = \mathbf{if}\ keyboard \neq noKB\ \mathbf{then}\ keyboard\ \mathbf{else}\ currentKeyedData$

▷ See: *Keyboard* (p. 24), *RealWorld* (p. 18), *present* (p. 8), *noKB* (p. 16)

As a result of polling the time and door the alarm may become raised or cleared and the latch locked or unlocked. Both of these events should be recorded in the audit. The opening and shutting of the door is also audited (auditing is defined later in the specification).

So the overall poll operation is obtained by combining all the individual polling actions.

If the user is currently being invited to enter the enclave on the display and the door becomes latched then the display will change to indicate that the system is no longer offering entry.

We assume that while polling occurs the *RealWorld* does not change.

TISPoll

$$
\begin{array}{l}
\Delta IDStation \\
\Xi RealWorld \\
\\
PollTime \\
PollDoor \\
PollUserToken \\
PollAdminToken \\
PollFinger \\
PollFloppy \\
PollKeyboard \\
LogChange \\
\\
\Xi Config \\
\Xi KeyStore \\
\Xi Admin \\
\Xi Stats \\
\Xi Internal \\
\hline
currentScreen' = currentScreen \\
\\
currentDisplay = doorUnlocked \land currentLatch' = locked \\
\quad \land (status \neq waitingRemoveTokenFail \land currentDisplay' = welcome \\
\qquad \lor status = waitingRemoveTokenFail \land currentDisplay' = removeToken) \\
\lor \neg (currentDisplay = doorUnlocked \land currentLatch' = locked) \\
\quad \land currentDisplay' = currentDisplay
\end{array}
$$

▷ See: *IDStation* (p. 25), *RealWorld* (p. 18), *PollTime* (p. 27), *PollDoor* (p. 27), *PollUserToken* (p. 28), *PollAdminToken* (p. 28), *PollFinger* (p. 28), *PollFloppy* (p. 28), *PollKeyboard* (p. 28), *Config* (p. 19), *KeyStore* (p. 21), *Admin* (p. 22), *Stats* (p. 21), *Internal* (p. 24), *doorUnlocked* (p. 15), *locked* (p. 15), *waitingRemoveTokenFail* (p. 24), *welcome* (p. 15)

4.3 The ID Station changes the world

4.3.1 Periodic Updates

We consider the process of updating the real world with the current internal representation, one variable at a time.

UpdateLatch

$$
\begin{array}{l}
\Xi DoorLatchAlarm \\
RealWorldChanges \\
\hline
latch' = currentLatch
\end{array}
$$

▷ See: *DoorLatchAlarm* (p. 22), *RealWorldChanges* (p. 27)

UpdateAlarm

$\Xi DoorLatchAlarm$
AuditLog
RealWorldChanges

$alarm' = alarming \Leftrightarrow doorAlarm = alarming \lor auditAlarm = alarming$

▷ See: *DoorLatchAlarm* (p. 22), *AuditLog* (p. 20), *RealWorldChanges* (p. 27), *alarming* (p. 15)

UpdateDisplay

$\Delta IDStation$
RealWorldChanges

$display' = currentDisplay$
$currentDisplay' = currentDisplay$

▷ See: *IDStation* (p. 25), *RealWorldChanges* (p. 27)

Configuration Data is only displayed if the security officer is present. System statistics are only displayed if an administrator is present.

UpdateScreen

$\Delta IDStation$
$\Xi Admin$
RealWorldChanges

$screen'.screenMsg = currentScreen.screenMsg$
$screen'.screenConfig = \textbf{if } the\ rolePresent = securityOfficer\ \textbf{then } currentScreen.screenConfig\ \textbf{else } clear$
$screen'.screenStats = \textbf{if } rolePresent \neq nil\ \textbf{then } currentScreen.screenStats\ \textbf{else } clear$

▷ See: *IDStation* (p. 25), *Admin* (p. 22), *RealWorldChanges* (p. 27), *the* (p. 8), *securityOfficer* (p. 9), *clear* (p. 16), *nil* (p. 8)

All these can be combined, along with no change in the remaining real world variables, to represent the regular updating of the world.

When updates to the real world occur it is possible that interfacing with external devices will result in a system fault that is audited. Not other aspects of TIS will change during updates of the real world.

FS.Interface.TISEarlyUpdate	
ScGainInitial.Suc.Locked	*FAU_ARP.1.1*
ScGainRepeat.Suc.Locked	*FAU_SAA.1.1*
ScUnlock.Suc.Locked	

The alarm and the door latch will need to be updated as soon as possible after polling the real world, this ensures that the system is kept secure.

$$
\begin{aligned}
TISEarlyUpdate \mathrel{\widehat=}\ & UpdateLatch \land UpdateAlarm \\
& \land [RealWorldChanges \mid screen' = screen \land display' = display] \\
& \land [\Delta IDStation \mid currentDisplay = currentDisplay'] \\
& \land \Xi UserToken \land \Xi AdminToken \land \Xi Finger \land \Xi Floppy \land \\
& \Xi Keyboard \land \Xi Config \land \Xi Stats \\
& \land \Xi KeyStore \land \Xi Admin \land \Xi Internal \\
& \land (AddElementsToLog \lor \Xi AuditLog)
\end{aligned}
$$

▷ See: *UpdateLatch* (p. 29), *UpdateAlarm* (p. 29), *RealWorldChanges* (p. 27), *IDStation* (p. 25), *UserToken* (p. 23), *AdminToken* (p. 23), *Finger* (p. 23), *Floppy* (p. 23), *Keyboard* (p. 24), *Config* (p. 19), *Stats* (p. 21), *KeyStore* (p. 21), *Admin* (p. 22), *Internal* (p. 24), *AuditLog* (p. 20)

FS.Interface.TISUpdate	
ScGainInitial.Suc.Locked	*FAU_SAA.1.2*
ScGainRepeat.Suc.Locked	*SFP.DAC*
ScUnlock.Suc.Locked	*FMT_MSA.1.1*
FAU_ARP.1.1	*FMT_SMR.2.2*
FAU_SAA.1.1	*FMT_SAE.1.1*

The alarm, door latch, display and TIS screen will be updated after performing any calculations.

$$\begin{aligned} TISUpdate \mathrel{\widehat=} & \; UpdateLatch \land UpdateAlarm \land UpdateDisplay \land UpdateScreen \\ & \land \Xi UserToken \land \Xi AdminToken \land \Xi Finger \land \Xi Floppy \land \\ & \Xi Keyboard \land \Xi Config \land \Xi Stats \\ & \land \Xi KeyStore \land \Xi Admin \land \Xi Internal \\ & \land (AddElementsToLog \lor \Xi AuditLog) \end{aligned}$$

▷ See: *UpdateLatch* (p. 29), *UpdateAlarm* (p. 29), *UpdateDisplay* (p. 30), *UpdateScreen* (p. 30), *UserToken* (p. 23), *AdminToken* (p. 23), *Finger* (p. 23), *Floppy* (p. 23), *Keyboard* (p. 24), *Config* (p. 19), *Stats* (p. 21), *KeyStore* (p. 21), *Admin* (p. 22), *Internal* (p. 24), *AuditLog* (p. 20)

4.3.2 Updating the user Token

FS.Interface.UpdateToken

We have a further operation, which writes to the User Token only. We treat this separately because we expect to update the other devices regularly and frequently, but we will only be updating the User Token when we have something to write.

UpdateUserToken

$\Xi IDStation$
RealWorldChanges

$\Xi TISControlledRealWorld$

$userToken' = currentUserToken$

▷ See: *IDStation* (p. 25), *RealWorldChanges* (p. 27), *TISControlledRealWorld* (p. 17)

4.3.3 Updating the Floppy

FS.Interface.UpdateFloppy

We have an operation which writes to the Floppy only. We will only be updating the Floppy disk when we have something to write.

UpdateFloppy

$\Delta IDStation$
RealWorldChanges

$\Xi UserToken$
$\Xi AdminToken$
$\Xi Finger$
$\Xi DoorLatchAlarm$
$\Xi Keyboard$
$\Xi Config$
$\Xi Stats$
$\Xi KeyStore$
$\Xi Admin$
$\Xi AuditLog$
$\Xi Internal$

$\Xi TISControlledRealWorld$

$floppy' = writtenFloppy$

$currentFloppy' = badFloppy$
$floppyPresence' = floppyPresence$
$currentDisplay' = currentDisplay$
$currentScreen' = currentScreen$

▷ See: *IDStation* (p. 25), *RealWorldChanges* (p. 27), *UserToken* (p. 23), *AdminToken* (p. 23), *Finger* (p. 23), *DoorLatchAlarm* (p. 22), *Keyboard* (p. 24), *Config* (p. 19), *Stats* (p. 21), *KeyStore* (p. 21), *Admin* (p. 22), *AuditLog* (p. 20), *Internal* (p. 24), *TISControlledRealWorld* (p. 17), *badFloppy* (p. 16)

▷ Having written the floppy we can assume nothing about the *currentFloppy* until we next poll. We do not know what data is on the floppy as it may have been corrupted during the write. This ensures that the readback we do is forced to be effective.

5 INTERNAL OPERATIONS

In this section we present a number of operations performed internally by the TIS. These operations are combined to create the operations available to the user.

5.1 Updating the Audit Log

5.1.1 Adding elements to the Log

FS.AuditLog.AddElementsToLog	
ScGeneral.Fail.Audit	*ScLogOn.Fail.AuditPreserve*
ScGainInitial.Fail.AuditPreserve	*ScLogOff.Fail.AuditPreserve*
ScProhibitInitial.Fail.AuditPreserve	*FAU_ARP.1.1*
ScGainRepeat.Fail.AuditPreserve	*FAU_SAA.1.1*
ScStart.Fail.AuditPreserve	*FAU_SAA.1.2*
ScShutdown.Fail.AuditPreserve	*FAU_STG.2.3*
ScConfig.Fail.AuditPreserve	*FAU_STG.4.1*
ScUnlock.Fail.AuditPreserve	

When we add a set of entries to the log, either there is sufficient room in the log for the new entries, in which case the new entries are added to the log, or there is insufficient room in the log to add the new entries and the oldest part of the log is truncated to make room for the new log entries. We don't specify here how much of the log is truncated although it is likely to be sufficient to continue adding some data without further truncations. If the log is truncated or is close to its maximum size, an alarm is raised to notify the administrator that the log is full.

AddElementsToLog

$$
\begin{array}{l}
Config \\
\Delta AuditLog \\
\hline
\exists\, newElements : \mathbb{F}_1\, Audit \bullet \\
\quad oldestLogTime\, newElements \geq newestLogTime\, auditLog \\
\quad \land\ (auditLog' = auditLog \cup newElements \\
\qquad \land\ (sizeLog\, auditLog' < alarmThresholdSize \land auditAlarm' = auditAlarm \\
\qquad\quad \lor\ sizeLog\, auditLog' \geq alarmThresholdSize \land auditAlarm' = alarming) \\
\qquad\qquad \lor \\
\quad sizeLog\, auditLog + sizeLog\, newElements > minPreservedLogSize \\
\qquad \land\ (\exists\, oldElements : \mathbb{F}\, Audit \bullet \\
\qquad\quad oldElements \cup auditLog' = auditLog \cup newElements \\
\qquad\quad \land\ oldestLogTime\, auditLog' \geq newestLogTime\, oldElements) \\
\qquad\quad \land\ sizeLog\, auditLog' \geq minPreservedLogSize \\
\qquad\quad \land\ auditAlarm' = alarming)
\end{array}
$$

▷ See: *Config* (p. 19), *AuditLog* (p. 20), *oldestLogTime* (p. 20), *alarming* (p. 15)

▷ We make an assuption that all data added to the log is no older than the data already in the log.

▷ This operation is non-deterministic when the addition of the set of *newElements* will make the log larger than the *minPreservedLogSize*. At this point the log may, or may not be truncated.

▷ If the configuration data changes it is possible that the *minPreservedLogSize* becomes larger or smaller, any new value for this configurable item will not take effect until configuration is complete.

5.1.2 Archiving the Log

FS.AuditLog.ArchiveLog

When we archive the log an audit element is added to the log and an archive is generated which can be written to floppy.

This activity does not clear the log since a check will be made to ensure the archive was successful before clearing the log.

ArchiveLog

$$
\begin{array}{l}
Config \\
\Delta AuditLog \\
archive : \mathbb{F}\ Audit \\
\hline
\exists\ notArchived, newElements : \mathbb{F}\ Audit \bullet \\
\quad archive \subseteq auditLog \cup newElements \\
\quad \land\ auditLog' \subseteq archive \cup notArchived \\
\quad \land\ newestLogTime\ archive \leq oldestLogTime\ notArchived \\
\quad \land\ AddElementsToLog
\end{array}
$$

▷ See: *Config* (p. 19), *AuditLog* (p. 20), *oldestLogTime* (p. 20), *AddElementsToLog* (p. 33)

▷ The explicit contraints on this schema define the component of the audit log that will be the *archive*. The constraints ensure that the *archive* includes the oldest elements and has no gaps in it.

▷ This operation is used in the total operation that writes the archive log to floppy. *archive* is the audit log that is written to floppy.

▷ The *archive* only contains some of the final log. The part of the log that is not archived is represented by *notArchived*.

5.1.3 Clearing the Log

FS.AuditLog.ClearLog
FAU_ARP.1.1

The log should only be cleared if it can be verified that an archive has been created of the data that is about to be cleared.

When the log is cleared the component that has been archived is eliminated from the log. There may still be some elements in the log, these will have been added since the archive. Where the log has overflowed since the time of the archive the archive may contain entries older than those in the log.

If the log is cleared successfully then the *auditAlarm* is cancelled (provided that the size of the audit log is not larger than the alarm threshold size).

ClearLog

$$
\begin{array}{l}
\mathit{Config} \\
\Delta \mathit{AuditLog} \\
\mathit{archive} : \mathbb{F}\ \mathit{Audit} \\
\hline
(\exists\, \mathit{sinceArchive}, \mathit{lostSinceArchive} : \mathbb{F}\ \mathit{Audit} \bullet \\
\quad \mathit{archive} \cup \mathit{sinceArchive} = \mathit{lostSinceArchive} \cup \mathit{auditLog} \\
\quad \land\ \mathit{oldestLogTime}\ \mathit{sinceArchive} \geq \mathit{newestLogTime}\ \mathit{archive} \\
\quad \land\ \mathit{newestLogTime}\ \mathit{lostSinceArchive} \leq \mathit{oldestLogTime}\ \mathit{auditLog} \\
\quad \land\ \mathit{auditLog}' = \mathit{sinceArchive}) \\
\\
(\mathit{sizeLog}\ \mathit{auditLog}' < \mathit{alarmThresholdSize} \land \mathit{auditAlarm}' = \mathit{silent} \\
\quad \lor\ \mathit{sizeLog}\ \mathit{auditLog}' \geq \mathit{alarmThresholdSize} \land \mathit{auditAlarm}' = \mathit{alarming})
\end{array}
$$

▷ See: *Config* (p. 19), *AuditLog* (p. 20), *oldestLogTime* (p. 20), *silent* (p. 15), *alarming* (p. 15)

▷ This operation is not total, it will only be used to construct a total operation that makes *archive* the value read back successfully from the floppy. Thus *archive* will have been the whole audit log at some point in the past.

5.1.4 Auditing Changes

FS.AuditLog.LogChange	
ScGainInitial.Suc.Audit	*FAU_ARP.1.1*
ScProhibitInitial.Suc.Audit	*FAU_SAA.1.1*
ScGainRepeat.Suc.Audit	*FAU_SAA.1.2*
ScUnlock.Suc.Audit	

TIS adds audit entries whenever any of the following changes occurs:

- The door is opened or closed.
- The door is latched or unlatched.
- The alarm starts alarming or becomes silenced.
- The audit alarm starts alarming or becomes silenced.
- The text displayed on the display changes.
- The text displayed on the screen changes.

AuditDoor

$$
\begin{array}{l}
\Delta \mathit{DoorLatchAlarm} \\
\mathit{AddElementsToLog} \\
\hline
\mathit{currentDoor} \neq \mathit{currentDoor}'
\end{array}
$$

▷ See: *DoorLatchAlarm* (p. 22), *AddElementsToLog* (p. 33)

AuditLatch

$$
\begin{array}{l}
\Delta \mathit{DoorLatchAlarm} \\
\mathit{AddElementsToLog} \\
\hline
\mathit{currentLatch}' \neq \mathit{currentLatch}
\end{array}
$$

▷ See: *DoorLatchAlarm* (p. 22), *AddElementsToLog* (p. 33)

AuditAlarm

$$\Delta DoorLatchAlarm$$
$$AddElementsToLog$$
$$doorAlarm \neq doorAlarm'$$

▷ See: *DoorLatchAlarm* (p. 22), *AddElementsToLog* (p. 33)

AuditLogAlarm

$$AddElementsToLog$$
$$auditAlarm \neq auditAlarm'$$

▷ See: *AddElementsToLog* (p. 33)

AuditDisplay

$$AddElementsToLog$$
$$\Delta IDStation$$
$$currentDisplay' \neq currentDisplay$$

▷ See: *AddElementsToLog* (p. 33), *IDStation* (p. 25)

AuditScreen

$$\Delta IDStation$$
$$AddElementsToLog$$
$$currentScreen'.screenMsg \neq currentScreen.screenMsg$$

▷ See: *IDStation* (p. 25), *AddElementsToLog* (p. 33)

If none of these changes occur then the audit log may still be updated due to the operation being executed; if no operation driven events occur it will not change.

NoChange

$$\Delta IDStation$$
$$currentDoor = currentDoor'$$
$$currentLatch' = currentLatch$$
$$doorAlarm = doorAlarm'$$
$$auditAlarm = auditAlarm'$$
$$currentDisplay' = currentDisplay$$
$$currentScreen'.screenMsg = currentScreen.screenMsg$$
$$AddElementsToLog \lor \Xi AuditLog$$

▷ See: *IDStation* (p. 25), *AddElementsToLog* (p. 33), *AuditLog* (p. 20)

▷ This is a very weak statement in the specification, because we are postponing elaboration of *Audit* elements until the design.

$$LogChange \mathrel{\widehat=} AuditAlarm \lor AuditLatch \lor AuditDoor \lor AuditLogAlarm \lor AuditScreen \lor AuditDisplay \lor NoChange$$

▷ See: *AuditAlarm* (p. 36), *AuditLatch* (p. 35), *AuditDoor* (p. 35), *AuditLogAlarm* (p. 36), *AuditScreen* (p. 36), *AuditDisplay* (p. 36), *NoChange* (p. 36)

5.2 Updating System Statistics

FS.Stats.Update

System statistics are updated as actions that are being monitored for the statistics occur.

We provide operations to increment the count of each of the events being monitored.

AddSuccessfulEntryToStats

$$\Delta Stats$$

$$failEntry' = failEntry$$
$$successEntry' = successEntry + 1$$
$$failBio' = failBio$$
$$successBio' = successBio$$

▷ See: *Stats* (p. 21)

AddFailedEntryToStats

$$\Delta Stats$$

$$failEntry' = failEntry + 1$$
$$successEntry' = successEntry$$
$$failBio' = failBio$$
$$successBio' = successBio$$

▷ See: *Stats* (p. 21)

AddSuccessfulBioCheckToStats

$$\Delta Stats$$

$$failEntry' = failEntry$$
$$successEntry' = successEntry$$
$$failBio' = failBio$$
$$successBio' = successBio + 1$$

▷ See: *Stats* (p. 21)

AddFailedBioCheckToStats

$$\Delta Stats$$

$$failEntry' = failEntry$$
$$successEntry' = successEntry$$
$$failBio' = failBio + 1$$
$$successBio' = successBio$$

▷ See: *Stats* (p. 21)

5.3 Operating the Door

FS.Door.UnlockDoor

The door is unlatched by updating the timeouts on the door latch and alarm.

UnlockDoor

$\Delta DoorLatchAlarm$
Config

$latchTimeout' = currentTime + latchUnlockDuration$
$alarmTimeout' = currentTime + latchUnlockDuration + alarmSilentDuration$
$currentTime' = currentTime$
$currentDoor' = currentDoor$

▷ See: *DoorLatchAlarm* (p. 22), *Config* (p. 19)

FS.Door.LockDoor

The door is explicitly latched and timeouts on the door latch and alarm are reset. Resetting the timeouts to the current time will ensure that the door will be latched directly and the alarm sound if there is a breach of security.

LockDoor

$\Delta DoorLatchAlarm$

$currentLatch' = locked$
$latchTimeout' = currentTime$
$alarmTimeout' = currentTime$
$currentTime' = currentTime$
$currentDoor' = currentDoor$

▷ See: *DoorLatchAlarm* (p. 22), *locked* (p. 15)

5.4 Certificate Operations

5.4.1 Validating Certificates

FS.Certificate.CertificateOK

When a certificate is checked in the context of a key store it is only acceptable if the certificate issuer is known to the key store and the signature can be verified by the key store.

A certificate must have been issued by a known issuer.

CertIssuerKnown

KeyStore
Certificate

$id.issuer \in \mathrm{dom}\, issuerKey$

▷ See: *KeyStore* (p. 21), *Certificate* (p. 10)

A certificate must have been signed by the issuer.

CertOK

$$\begin{array}{l} \textit{CertIssuerKnown} \\ \hline \textit{issuerKey}(\textit{id}.\textit{issuer}) \in \textit{isValidatedBy} \end{array}$$

▷ See: *CertIssuerKnown* (p. 38)

FS.Certificate.AuthCertificateOK

In addition the Authorisation certificate must have been issued by this ID station; we make the assumption that a single ID station protects an enclave.

CertIssuerIsThisTIS

$$\begin{array}{l} \textit{KeyStore} \\ \textit{Certificate} \\ \hline \textit{ownName} \neq \textit{nil} \\ \textit{id}.\textit{issuer} = \textit{the ownName} \end{array}$$

▷ See: *KeyStore* (p. 21), *Certificate* (p. 10), *nil* (p. 8), *the* (p. 8)

$$\textit{AuthCertOK} \mathrel{\widehat{=}} \textit{CertIssuerIsThisTIS} \land \textit{CertOK}$$

▷ See: *CertIssuerIsThisTIS* (p. 39), *CertOK* (p. 39)

5.4.2 Generating Authorisation Certificates

FS.Certificate.NewAuthCert

FDP_UIT.1.1 *FIA_UAU.3.2*
FDP_UIT.1.2

An authorisation certificate can be constructed using information from a valid token and the current configuration of TIS. TIS can only generate the authorisation certificate if it has its own key to perform the signing with; this is modelled as the TIS knowing its own name.

NewAuthCert

$$\begin{array}{l} \textit{ValidToken} \\ \textit{KeyStore} \\ \textit{Config} \\ \textit{newAuthCert} : \textit{AuthCert} \\ \textit{currentTime} : \textit{TIME} \\ \hline \textit{ownName} \neq \textit{nil} \\ \\ \textit{newAuthCert}.\textit{id}.\textit{issuer} = \textit{the ownName} \\ \textit{newAuthCert}.\textit{validityPeriod} = \textit{authPeriod privCert}.\textit{role currentTime} \\ \textit{newAuthCert}.\textit{baseCertId} = \textit{idCert}.\textit{id} \\ \textit{newAuthCert}.\textit{tokenID} = \textit{tokenID} \\ \textit{newAuthCert}.\textit{role} = \textit{privCert}.\textit{role} \\ \textit{newAuthCert}.\textit{clearance} = \textit{minClearance}(\textit{enclaveClearance}, \textit{privCert}.\textit{clearance}) \\ \textit{newAuthCert}.\textit{isValidatedBy} = \{\, \textit{issuerKey}(\textit{the ownName}) \,\} \end{array}$$

▷ See: *ValidToken* (p. 13), *KeyStore* (p. 21), *Config* (p. 19), *AuthCert* (p. 12), *TIME* (p. 8), *nil* (p. 8), *the* (p. 8), *minClearance* (p. 9)

5.4.3 Adding Authorisation Certificates to User Token

FS.UserToken.AddAuthCertToUserToken

If a valid user token is present in the system then an authorisation certificate can be added to it.

AddAuthCertToUserToken

$$
\begin{array}{l}
\Delta UserToken \\
KeyStore \\
Config \\
currentTime : TIME \\
\hline
userTokenPresence = present \\
currentUserToken \in \mathrm{ran}\, goodT \\
\exists\, ValidToken; ValidToken' \bullet \theta ValidToken = (goodT^{\sim} currentUserToken) \\
\quad \land\ \theta ValidToken' = (goodT^{\sim} currentUserToken') \\
\quad \land\ (\exists\, newAuthCert : AuthCert \bullet the\, authCert' = newAuthCert \land NewAuthCert) \\
\quad \land\ tokenID' = tokenID \\
\quad \land\ idCert' = idCert \\
\quad \land\ privCert' = privCert \\
\quad \land\ iandACert' = iandACert \\
userTokenPresence' = userTokenPresence
\end{array}
$$

▷ See: *UserToken* (p. 23), *KeyStore* (p. 21), *Config* (p. 19), *TIME* (p. 8), *present* (p. 8), *goodT* (p. 15), *ValidToken* (p. 13), *AuthCert* (p. 12), *the* (p. 8), *NewAuthCert* (p. 39)

5.5 Updating the Key Store

FS.KeyStore.UpdateKeyStore

The key store is updated using the supplied enrolment data to add issuers and their public keys.

UpdateKeyStore

$$
\begin{array}{l}
\Delta KeyStore \\
ValidEnrol \\
\hline
the\, ownName' = idStationCert.subject \\
issuerKey' = issuerKey \oplus \{c : issuerCerts \bullet c.subject \mapsto c.subjectPubK\} \\
\quad \oplus \{the\, ownName \mapsto idStationCert.subjectPubK\}
\end{array}
$$

▷ See: *KeyStore* (p. 21), *ValidEnrol* (p. 14), *the* (p. 8)

▷ This operation uses union and override so that it can be used to add issuers as well as initial enrolment.

The enrolment data will always be supplied on a floppy disk.

UpdateKeyStoreFromFloppy

Δ*KeyStore*
Floppy

---

currentFloppy $\in \mathrm{ran}$ *enrolmentFile*
$(\exists$ *ValidEnrol* $\bullet$ θ*ValidEnrol* $=$ *enrolmentFile*$^{\sim}$*currentFloppy*
$\quad \wedge$ *UpdateKeyStore*$)$

▷ See: *KeyStore* (p. 21), *Floppy* (p. 23), *enrolmentFile* (p. 16), *ValidEnrol* (p. 14), *UpdateKeyStore* (p. 40)

5.6 Administrator Changes

An administrator may log on to the TIS console, logoff, or start an operation.

5.6.1 Logon Administrator

FS.Admin.AdminLogon

An administrator can only log on if there is no-one currently logged on.

AdminLogon

Δ*Admin*
AdminToken

---

rolePresent $=$ *nil*

$\exists$ *ValidToken* $\bullet$
$\quad$ (*goodT*(θ*ValidToken*) $=$ *currentAdminToken*
$\quad \wedge$ *the rolePresent*$'$ $=$ (*the authCert*).*role*)

currentAdminOp$'$ $=$ *nil*

▷ See: *Admin* (p. 22), *AdminToken* (p. 23), *nil* (p. 8), *ValidToken* (p. 13), *goodT* (p. 15), *the* (p. 8)

5.6.2 Logout Administrator

FS.Admin.AdminLogout

An adminstrator, who is currently logged on can always log off. This will terminate the current operation.

AdminLogout

Δ*Admin*

---

rolePresent $\neq$ *nil*

rolePresent$'$ $=$ *nil*
currentAdminOp$'$ $=$ *nil*

▷ See: *Admin* (p. 22), *nil* (p. 8)

5.6.3 Administrator Starts Operation

FS.Admin.AdminStartOp

An adminstrator, who is currently logged on, can start any of the operations that he is allowed to perform. An operation can only be started if there is no operation currently in progress.

AdminStartOp

$\Delta Admin$
$Keyboard$

$rolePresent \neq nil$
$currentAdminOp = nil$
$currentKeyedData \in keyedOps(\!|availableOps|\!)$

$rolePresent' = rolePresent$
$the\, currentAdminOp' = keyedOps^{\sim} currentKeyedData$

▷ See: *Admin* (p. 22), *Keyboard* (p. 24), *nil* (p. 8), *keyedOps* (p. 16), *the* (p. 8)

5.6.4 Administrator Finishes Operation

FS.Admin.AdminFinishOp

An adminstrator, who is currently logged on, can finish an operation.

AdminFinishOp

$\Delta Admin$

$rolePresent \neq nil$
$currentAdminOp \neq nil$

$rolePresent' = rolePresent$
$currentAdminOp' = nil$

▷ See: *Admin* (p. 22), *nil* (p. 8)

6 THE USER ENTRY OPERATION

This operation is a multi-stage operation and will be presented as a number of operations with preconditions on the internal *status*. The state transition diagram for user authentication and entry is given in Figure 6.1. Before user authentication and entry the system is in the *quiescent* state, on completion of the user authentication and entry the system will return the to *quiescent* state.

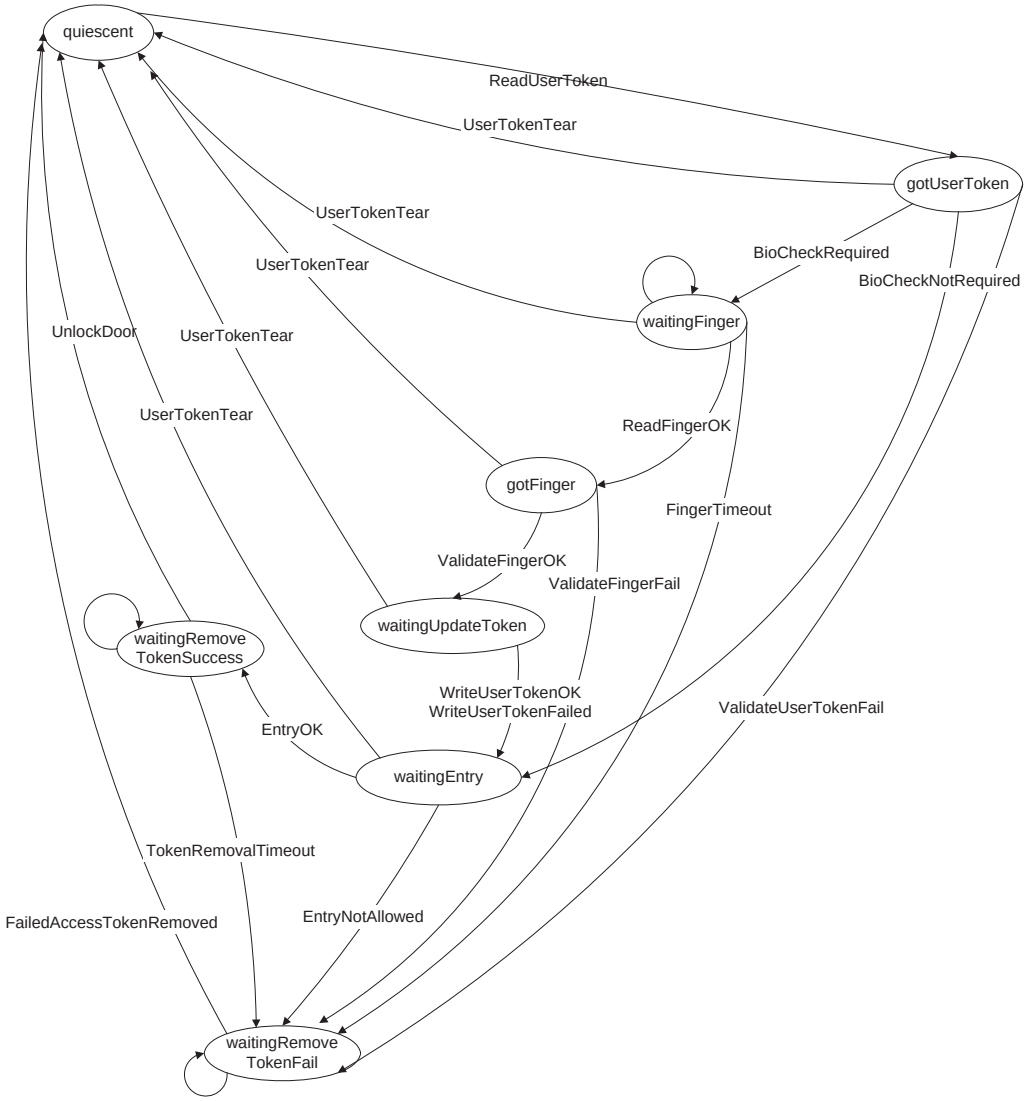

Figure 6.1: User Authentication and Entry state transitions

The process of user authentication and entry follows the following stages:

- Before any user attempts access, the system is *quiescent*.
- Once the token has been inserted and the information read off, the status moves to *gotUserToken*, waiting for the system to validate the token.

- Once the token has been successfully validated the status moves to *waitingFinger*, waiting for the user to give a fingerprint.
- Once the fingerprint has been read, the status moves to *gotFinger*, waiting for the system to validate the fingerprint.
- Once a fingerprint has been successfully validated, the status moves to *waitingUpdateToken*, waiting to write the Auth Cert to the token.
- Once the Auth Cert has been written, the status moves to *waitingEntry*, where it determines whether the role has current entry privileges.
- If the role has current entry privileges the status moves to *waitingTokenRemoveSuccess*, where the system waits for the token to be removed.
- Once the token has been removed the latch will be unlocked if the role has current access privileges to the enclave and the ID Station will return to *quiescent*.

In the case of a failure in the user validation process the status moves to *waitingRemoveTokenFail*, waiting until the token has been removed before returning to a *quiescent* state.

This specification separates opening the door from having a valid Auth Certificate. It is possible for a role to be entitled to enter the enclave but not use the workstations (for example such clearence might be given to a buildings maintenance engineer). It is likely that TIS configurations will ensure that having a valid Auth Certificate will guarantee that entry to the enclave is permitted.

FS.Enclave.ResetScreenMessage

The message displayed on the screen will indicate that the system is busy while a user entry is in progress that blocks administrator activity. Once the user entry activity becomes non-blocking then an appropriate message is displayed on the screen.

ResetScreenMessage

$$
\begin{array}{l}
\Delta Internal \\
\Delta Admin \\
currentScreen, currentScreen' : Screen \\
\hline
status' \notin \{quiescent, waitingRemoveTokenFail\} \\
\quad \land\ currentScreen'.screenMsg = busy \\
\lor \\
status' \in \{quiescent, waitingRemoveTokenFail\} \\
\quad \land\ (enclaveStatus' = enclaveQuiescent \land rolePresent' = nil \\
\qquad \land\ currentScreen'.screenMsg = welcomeAdmin \\
\qquad \lor\ enclaveStatus' = enclaveQuiescent \land rolePresent' \neq nil \\
\qquad \land\ currentScreen'.screenMsg = requestAdminOp \\
\qquad \lor\ enclaveStatus' = waitingRemoveAdminTokenFail \\
\qquad \land\ currentScreen'.screenMsg = removeAdminToken \\
\qquad \lor\ enclaveStatus' \notin \{enclaveQuiescent, waitingRemoveAdminTokenFail\} \\
\qquad \land\ currentScreen'.screenMsg = currentScreen.screenMsg)
\end{array}
$$

▷ See: *Internal* (p. 24), *Admin* (p. 22), *Screen* (p. 16), *quiescent* (p. 24), *waitingRemoveTokenFail* (p. 24), *busy* (p. 16), *nil* (p. 8), *welcomeAdmin* (p. 16), *waitingRemoveAdminTokenFail* (p. 24), *removeAdminToken* (p. 16)

The user entry operation leaves much of the *IDStation* state unchanged. The context of this operation is summarised:

UserEntryContext

$\Delta IDStation$
RealWorldChanges

$\Xi Config$
$\Xi AdminToken$
$\Xi KeyStore$
$\Xi Admin$
$\Xi Keyboard$
$\Xi Floppy$
$\Xi Finger$

$\Xi TISControlledRealWorld$

ResetScreenMessage

$enclaveStatus' = enclaveStatus$
$status \neq waitingEntry \Rightarrow tokenRemovalTimeout' = tokenRemovalTimeout$

▷ See: *IDStation* (p. 25), *RealWorldChanges* (p. 27), *Config* (p. 19), *AdminToken* (p. 23), *KeyStore* (p. 21), *Admin* (p. 22), *Keyboard* (p. 24), *Floppy* (p. 23), *Finger* (p. 23), *TISControlledRealWorld* (p. 17), *ResetScreenMessage* (p. 44), *waitingEntry* (p. 24)

▷ The following state components may change *UserToken*, *DoorLatchAlarm*, *Stats*, *Internal* and *AuditLog*.

▷ The components of the real world controlled by TIS remain unchanged.

▷ The *tokenRemovalTimeout* is only updated if the current status is *waitingEntry*.

Each of the following sub-operations is performed within the above context.

6.1 User Token Tears

FS.UserEntry.UserTokenTorn

ScGainInitial.Suc.Audit	*ScProhibitInitial.Fail.ReadCard*
ScGainInitial.Fail.ReadCard	*ScGainRepeat.Suc.Audit*
ScProhibitInitial.Suc.Audit	

During the operation the user may tear his token from the reader prematurely. There are a number of internal states during which token removal is deamed erroneous.

If the user tears the Token out before the operation is complete then the operation is terminated unsuccessfully.

UserTokenTorn

UserEntryContext

$\Xi UserToken$
$\Xi DoorLatchAlarm$
AddFailedEntryToStats

AddElementsToLog

$status \in \{gotUserToken, waitingUpdateToken, waitingFinger, gotFinger, waitingEntry\}$
$userTokenPresence = absent$

$currentDisplay' = welcome$
$status' = quiescent$

▷ See: *UserEntryContext* (p. 44), *UserToken* (p. 23), *DoorLatchAlarm* (p. 22), *AddFailedEntryToStats* (p. 37), *AddElementsToLog* (p. 33), *waitingUpdateToken* (p. 24), *waitingFinger* (p. 24), *gotFinger* (p. 24), *waitingEntry* (p. 24), *absent* (p. 8), *welcome* (p. 15), *quiescent* (p. 24)

6.2 Reading the User Token

FS.UserEntry.TISReadUserToken	
ScGainInitial.Ass.Quiescent	*ScProhibitInitial.Suc.Audit*
ScGainInitial.Suc.Audit	*ScGainRepeat.Ass.Quiescent*
ScGainInitial.Con.NoInterleave	*ScGainRepeat.Suc.Audit*
ScProhibitInitial.Ass.Quiescent	*ScGainRepeat.Con.NoInterleave*
ScProhibitInitial.Con.NoInterleave	*FIA_UID.2.1*

The User Entry operation is initiated when TIS is in a *quiescent* state and detects the presence of a token in the user token reader (which resides outside the enclave).

A user entry operation may start while the *enclaveStatus* is quiescent (*enclaveQuiescent*) or the enclave is waiting for a failed admin token to be removed.

When the user token is first detected as present, its presence is audited and the internal status changes. No other aspects of the system are modified.

ReadUserToken

$$
\begin{array}{l}
UserEntryContext \\
\\
\Xi UserToken \\
\Xi DoorLatchAlarm \\
\Xi Stats \\
\\
AddElementsToLog \\
\hline
enclaveStatus \in \{enclaveQuiescent, waitingRemoveAdminTokenFail\} \\
\\
status = quiescent \\
userTokenPresence = present \\
\\
currentDisplay' = wait \\
status' = gotUserToken
\end{array}
$$

▷ See: *UserEntryContext* (p. 44), *UserToken* (p. 23), *DoorLatchAlarm* (p. 22), *Stats* (p. 21), *AddElementsToLog* (p. 33), *waitingRemoveAdminTokenFail* (p. 24), *quiescent* (p. 24), *present* (p. 8), *wait* (p. 15)

The operation to read the user token is as follows:

$$TISReadUserToken \mathrel{\widehat=} ReadUserToken$$

▷ See: *ReadUserToken* (p. 46)

6.3 Validating the User Token

Once TIS has read a user token it must validate the contents of that token.

A user token is valid for entry into the enclave, without the need for Biometric checks if the token contains an Authorisation certificate that cross-checks correctly with the Token ID and the ID certificate, is current and both the Authorisation certificate and ID certificate can be validated using the keys held in the TIS *KeyStore*.

UserTokenWithOKAuthCert

$$
\begin{array}{l}
KeyStore \\
UserToken \\
currentTime : TIME \\
\hline
currentUserToken \in \operatorname{ran} goodT \\
\exists\, TokenWithValidAuth \bullet \\
\quad (goodT(\theta TokenWithValidAuth) = currentUserToken \\
\quad \land\ currentTime \in (the\ authCert).validityPeriod \\
\quad \land\ (\exists\, IDCert \bullet \theta IDCert = idCert \land CertOK) \\
\quad \land\ (\exists\, AuthCert \bullet \theta AuthCert = the\ authCert \land AuthCertOK))
\end{array}
$$

▷ See: *KeyStore* (p. 21), *UserToken* (p. 23), *TIME* (p. 8), *goodT* (p. 15), *TokenWithValidAuth* (p. 13), *the* (p. 8), *IDCert* (p. 11), *CertOK* (p. 39), *AuthCert* (p. 12), *AuthCertOK* (p. 39)

A user token is valid for entry into the enclave if the token is consistent, current and the ID certificate, Privilege certificate and I&A certificate can be validated. This is regardless of the presence or state of the Authorisation certificate. However in this circumstance biometric checks will be required.

UserTokenOK

$$
\begin{array}{l}
KeyStore \\
UserToken \\
currentTime : TIME \\
\hline
currentUserToken \in \operatorname{ran} goodT \\
\exists\, CurrentToken \bullet \\
\quad (goodT(\theta ValidToken) = currentUserToken \\
\quad \land\ now = currentTime \\
\quad \land\ (\exists\, IDCert \bullet \theta IDCert = idCert \land CertOK) \\
\quad \land\ (\exists\, PrivCert \bullet \theta PrivCert = privCert \land CertOK) \\
\quad \land\ (\exists\, IandACert \bullet \theta IandACert = iandACert \land CertOK))
\end{array}
$$

▷ See: *KeyStore* (p. 21), *UserToken* (p. 23), *TIME* (p. 8), *goodT* (p. 15), *CurrentToken* (p. 13), *ValidToken* (p. 13), *IDCert* (p. 11), *CertOK* (p. 39), *PrivCert* (p. 12), *IandACert* (p. 12)

FS.UserEntry.BioCheckNotRequired

ScGainInitial.Ass.GoodAC	*FCO_NRO.2.3*
ScGainRepeat.Suc.Audit	*FDP_DAU.2.1*
FCO_NRO.2.1	*FDP_DAU.2.2*
FCO_NRO.2.1	

In the case where there is a valid Authorisation certificate the biometric checks are bypassed.

BioCheckNotRequired

$$
\begin{array}{l}
UserEntryContext \\
\\
\Xi UserToken \\
\Xi DoorLatchAlarm \\
\Xi Stats \\
\\
AddElementsToLog \\
\hline
status = gotUserToken \\
userTokenPresence = present \\
\\
UserTokenWithOKAuthCert \\
\\
status' = waitingEntry \\
currentDisplay' = wait
\end{array}
$$

▷ See: *UserEntryContext* (p. 44), *UserToken* (p. 23), *DoorLatchAlarm* (p. 22), *Stats* (p. 21), *AddElementsToLog* (p. 33), *present* (p. 8), *UserTokenWithOKAuthCert* (p. 46), *waitingEntry* (p. 24), *wait* (p. 15)

▷ The *userTokenValidElement* is the audit entry recording that the token has been succesfully validated.

▷ The *authCertValidElement* is the audit entry recording that the token has a valid authorisation certificate.

FS.UserEntry.BioCheckRequired	
ScGainInitial.Ass.ValidUser	*FCO_NRO.2.1*
ScGainInitial.Ass.PoorAC	*FCO_NRO.2.3*
ScGainInitial.Suc.Audit	*FDP_DAU.2.1*
FCO_NRO.2.1	*FDP_DAU.2.2*

The biometric checks are only required if the Authorisation Certificate is not present or not valid. In this case the remaining certificates on the card must be checked.

BioCheckRequired

$$
\begin{array}{l}
UserEntryContext \\
\\
\Xi UserToken \\
\Xi DoorLatchAlarm \\
\Xi Stats \\
\\
AddElementsToLog \\
\hline
status = gotUserToken \\
userTokenPresence = present \\
\\
\neg\ UserTokenWithOKAuthCert \land UserTokenOK \\
\\
currentDisplay' = insertFinger \\
status' = waitingFinger
\end{array}
$$

▷ See: *UserEntryContext* (p. 44), *UserToken* (p. 23), *DoorLatchAlarm* (p. 22), *Stats* (p. 21), *AddElementsToLog* (p. 33), *present* (p. 8), *UserTokenWithOKAuthCert* (p. 46), *UserTokenOK* (p. 47), *insertFinger* (p. 15), *waitingFinger* (p. 24)

$$ValidateUserTokenOK \mathrel{\widehat=} BioCheckRequired \lor BioCheckNotRequired$$

▷ See: *BioCheckRequired* (p. 48), *BioCheckNotRequired* (p. 47)

FS.UserEntry.ValidateUserTokenFail	
ScGainInitial.Fail.ReadCard	*ScProhibitInitial.Suc.Audit*
ScProhibitInitial.Ass.FalseUser	*ScProhibitInitial.Fail.ReadCard*
ScProhibitInitial.Ass.PoorAC	*ScGainRepeat.Fail.ReadCard*

There are lots of things that may go wrong with validation of the user token. In each case the system will terminate the operation unsuccessfully.

ValidateUserTokenFail

UserEntryContext

Ξ*UserToken*
Ξ*DoorLatchAlarm*
Ξ*Stats*

AddElementsToLog

---

status $=$ *gotUserToken*
userTokenPresence $=$ *present*

$\neg$ *UserTokenWithOKAuthCert* $\wedge$ $\neg$ *UserTokenOK*

currentDisplay$'$ $=$ *removeToken*
status$'$ $=$ *waitingRemoveTokenFail*

▷ See: *UserEntryContext* (p. 44), *UserToken* (p. 23), *DoorLatchAlarm* (p. 22), *Stats* (p. 21), *AddElementsToLog* (p. 33), *present* (p. 8), *UserTokenWithOKAuthCert* (p. 46), *UserTokenOK* (p. 47), *waitingRemoveTokenFail* (p. 24)

$$TISValidateUserToken \mathrel{\widehat=} ValidateUserTokenOK \lor ValidateUserTokenFail \lor [\, UserTokenTorn \mid status = gotUserToken \,]$$

▷ See: *ValidateUserTokenOK* (p. 48), *ValidateUserTokenFail* (p. 48), *UserTokenTorn* (p. 45)

6.4 Reading a fingerprint

FS.UserEntry.ReadFingerOK

ScGainInitial.Suc.Audit *ScProhibitInitial.Suc.Audit*

A finger will be read if the system is currently waiting for it and the user Token is in place.

ReadFingerOK

UserEntryContext

Ξ*DoorLatchAlarm*
Ξ*UserToken*
Ξ*Stats*

AddElementsToLog

---

status $=$ *waitingFinger*
fingerPresence $=$ *present*
userTokenPresence $=$ *present*

currentDisplay$'$ $=$ *wait*
status$'$ $=$ *gotFinger*

▷ See: *UserEntryContext* (p. 44), *DoorLatchAlarm* (p. 22), *UserToken* (p. 23), *Stats* (p. 21), *AddElementsToLog* (p. 33), *waitingFinger* (p. 24), *present* (p. 8), *wait* (p. 15), *gotFinger* (p. 24)

FS.UserEntry.NoFinger

If there is no finger present then either nothing happens, since we have not allowed sufficient attempts to get and validate a finger...

NoFinger

$\Xi IDStation$
RealWorldChanges

$\Xi TISControlledRealWorld$

$status = waitingFinger$
$fingerPresence = absent$
$userTokenPresence = present$

▷ See: *IDStation* (p. 25), *RealWorldChanges* (p. 27), *TISControlledRealWorld* (p. 17), *waitingFinger* (p. 24), *absent* (p. 8), *present* (p. 8)

FS.UserEntry.FingerTimeout	
ScGainInitial.Fail.Fingerprint	*ScProhibitInitial.Fail.Fingerprint*
ScProhibitInitial.Suc.Audit	

...or TIS may have tried to obtain a valid finger for too long, in which case the user is requested to remove the token and the operation is terminated unsuccessfully. Abstractly this decision is made non-deterministically.

FingerTimeout

UserEntryContext

$\Xi UserToken$
$\Xi DoorLatchAlarm$
$\Xi Stats$

AddElementsToLog

$status = waitingFinger$
$userTokenPresence = present$

$currentDisplay' = removeToken$
$status' = waitingRemoveTokenFail$

▷ See: *UserEntryContext* (p. 44), *UserToken* (p. 23), *DoorLatchAlarm* (p. 22), *Stats* (p. 21), *AddElementsToLog* (p. 33), *waitingFinger* (p. 24), *present* (p. 8), *waitingRemoveTokenFail* (p. 24)

$$TISReadFinger \mathrel{\widehat=} ReadFingerOK \lor FingerTimeout \lor NoFinger \lor [\, UserTokenTorn \mid status = waitingFinger \,]$$

▷ See: *ReadFingerOK* (p. 49), *FingerTimeout* (p. 50), *NoFinger* (p. 49), *UserTokenTorn* (p. 45), *waitingFinger* (p. 24)

6.5 Validating a fingerprint

FS.UserEntry.ValidateFingerOK	
ScGainInitial.Ass.ValidUser	*ScGainInitial.Suc.Audit*

A finger must match the template information extracted from the userToken for it to be considered acceptable. The match criterion is not modelled formally here athough it is necessary for the fingerprint to at least be good.

FingerOK

Finger
UserToken

$currentFinger \in \operatorname{ran} goodFP$

▷ See: *Finger* (p. 23), *UserToken* (p. 23), *goodFP* (p. 15)

Within this specification the fingerprint will non-deterministically match or not, assuming it is good.

The fingerprint being successfully validated is a prerequisite for generating an authorisation certificate and adding it to the user token. Validating the fingerprint is performed first.

ValidateFingerOK

UserEntryContext

Ξ*DoorLatchAlarm*
Ξ*UserToken*

AddSuccessfulBioCheckToStats
AddElementsToLog

$status = gotFinger$
$userTokenPresence = present$

FingerOK

$status' = waitingUpdateToken$
$currentDisplay' = wait$

▷ See: *UserEntryContext* (p. 44), *DoorLatchAlarm* (p. 22), *UserToken* (p. 23), *AddSuccessfulBioCheckToStats* (p. 37), *AddElementsToLog* (p. 33), *gotFinger* (p. 24), *present* (p. 8), *FingerOK* (p. 50), *waitingUpdateToken* (p. 24), *wait* (p. 15)

FS.UserEntry.ValidateFingerFail

ScGainInitial.Fail.Fingerprint *ScProhibitInitial.Suc.Audit*
ScProhibitInitial.Ass.FalseUser

If the fingerprint is not successfully validated the user is asked to remove their token and the entry attempt is terminated. The biometric check failure is recorded.

ValidateFingerFail

UserEntryContext

Ξ*UserToken*
Ξ*DoorLatchAlarm*

AddFailedBioCheckToStats
AddElementsToLog

$status = gotFinger$
$userTokenPresence = present$

$currentDisplay' = removeToken$
$status' = waitingRemoveTokenFail$

▷ See: *UserEntryContext* (p. 44), *UserToken* (p. 23), *DoorLatchAlarm* (p. 22), *AddFailedBioCheckToStats* (p. 37), *AddElementsToLog* (p. 33), *gotFinger* (p. 24), *present* (p. 8), *waitingRemoveTokenFail* (p. 24)

$$TISValidateFinger \mathrel{\widehat=} ValidateFingerOK \lor ValidateFingerFail \lor [\, UserTokenTorn \mid status = gotFinger \,]$$

▷ See: *ValidateFingerOK* (p. 51), *ValidateFingerFail* (p. 51), *UserTokenTorn* (p. 45), *gotFinger* (p. 24)

6.6 Writing the User Token

FS.UserEntry.WriteUserTokenOK

ScGainInitial.Suc.GoodAC *ScGainInitial.Suc.Audit*
ScGainInitial.Suc.PersistCerts

An attempt is made to write this certificate to the token. The write of the authorisation certificate may be successful...

WriteUserTokenOK

$$UserEntryContext$$

$$\Xi DoorLatchAlarm$$
$$\Xi Stats$$

$$AddElementsToLog$$
$$AddAuthCertToUserToken$$

$$status = waitingUpdateToken$$
$$userTokenPresence = present$$

$$status' = waitingEntry$$
$$currentDisplay' = wait$$

▷ See: *UserEntryContext* (p. 44), *DoorLatchAlarm* (p. 22), *Stats* (p. 21), *AddElementsToLog* (p. 33), *AddAuthCertToUserToken* (p. 40), *waitingUpdateToken* (p. 24), *present* (p. 8), *waitingEntry* (p. 24), *wait* (p. 15)

FS.UserEntry.WriteUserTokenFail

ScGainInitial.Fail.WriteCard

... or may fail. The failure case models circumstances where the TIS can detect the failure, through a write failure for instance. As there is no read back of the authorisation certificate we cannot guarantee that the audit log indicating a successful write means that the token contains the authorisation certificate. The user will still subsequently be admitted to the enclave if the conditions are correct.

WriteUserTokenFail

$$UserEntryContext$$

$$\Xi DoorLatchAlarm$$
$$\Xi Stats$$

$$AddElementsToLog$$
$$AddAuthCertToUserToken$$

$$status = waitingUpdateToken$$
$$userTokenPresence = present$$

$$status' = waitingEntry$$
$$currentDisplay' = tokenUpdateFailed$$

▷ See: *UserEntryContext* (p. 44), *DoorLatchAlarm* (p. 22), *Stats* (p. 21), *AddElementsToLog* (p. 33), *AddAuthCertToUserToken* (p. 40), *waitingUpdateToken* (p. 24), *present* (p. 8), *waitingEntry* (p. 24), *tokenUpdateFailed* (p. 15)

Abstractly, whether the authorisation certificate is successfully written or not is non-deterministic.

The failure will actually happen during the physical write to the token, during *UpdateUserToken*. However, as the operations *WriteUserToken* and *UpdateUserToken* are both used to build the atomic operation *TISWriteUserToken*, the non-deterministic failure still happens sometime within this atomic operation.

$$WriteUserToken \mathrel{\widehat=} WriteUserTokenOK \lor WriteUserTokenFail$$

▷ See: *WriteUserTokenOK* (p. 52), *WriteUserTokenFail* (p. 52)

$$TISWriteUserToken \mathrel{\widehat=} (WriteUserToken \fatsemi UpdateUserToken) \lor [\, UserTokenTorn \mid status = waitingUpdateToken \,]$$

▷ See: *WriteUserToken* (p. 53), *UpdateUserToken* (p. 31), *UserTokenTorn* (p. 45), *waitingUpdateToken* (p. 24)

6.7 Validating Entry

The door will only be unlocked if the current TIS configuration allows the user to enter the enclave at this time. It is likely that TIS configurations will ensure that having a valid Auth Certificate will guarantee that entry to the enclave is permitted, but such a constraint is not specified here.

TIS checks to ensure that the current configuration allows the user to enter the enclave:

UserAllowedEntry

$$\begin{array}{l} UserToken \\ Config \\ currentTime : TIME \\ \hline (\exists\, ValidToken \bullet \\ \quad goodT(\theta ValidToken) = currentUserToken \\ \quad \land\ currentTime \in entryPeriod\ privCert.role\ privCert.clearance.class) \\ \lor\ (\exists\, TokenWithValidAuth \bullet \\ \quad goodT(\theta TokenWithValidAuth) = currentUserToken \\ \quad \land\ currentTime \in entryPeriod\ (the\ authCert).role\ (the\ authCert).clearance.class) \end{array}$$

▷ See: *UserToken* (p. 23), *Config* (p. 19), *TIME* (p. 8), *ValidToken* (p. 13), *goodT* (p. 15), *TokenWithValidAuth* (p. 13), *the* (p. 8)

FS.UserEntry.EntryOK

Only if entry is permitted at the current time will the user be admitted to the enclave.

EntryOK

UserEntryContext

Ξ*DoorLatchAlarm*
Ξ*UserToken*
Ξ*Stats*

AddElementsToLog

$status = waitingEntry$
$userTokenPresence = present$

UserAllowedEntry

$currentDisplay' = openDoor$
$status' = waitingRemoveTokenSuccess$
$tokenRemovalTimeout' = currentTime + tokenRemovalDuration$

▷ See: *UserEntryContext* (p. 44), *DoorLatchAlarm* (p. 22), *UserToken* (p. 23), *Stats* (p. 21), *AddElementsToLog* (p. 33), *waitingEntry* (p. 24), *present* (p. 8), *UserAllowedEntry* (p. 53), *openDoor* (p. 15)

FS.UserEntry.EntryNotAllowed

If the user is not allowed entry at this time they will be requested to remove their token.

EntryNotAllowed

UserEntryContext

Ξ*DoorLatchAlarm*
Ξ*UserToken*
Ξ*Stats*

AddElementsToLog

$status = waitingEntry$
$userTokenPresence = present$

$\neg$ *UserAllowedEntry*

$currentDisplay' = removeToken$
$status' = waitingRemoveTokenFail$
$tokenRemovalTimeout' = tokenRemovalTimeout$

▷ See: *UserEntryContext* (p. 44), *DoorLatchAlarm* (p. 22), *UserToken* (p. 23), *Stats* (p. 21), *AddElementsToLog* (p. 33), *waitingEntry* (p. 24), *present* (p. 8), *UserAllowedEntry* (p. 53), *waitingRemoveTokenFail* (p. 24)

$$TISValidateEntry \mathrel{\widehat=} EntryOK \lor EntryNotAllowed \lor [\, UserTokenTorn \mid status = waitingEntry \,]$$

▷ See: *EntryOK* (p. 53), *EntryNotAllowed* (p. 54), *UserTokenTorn* (p. 45), *waitingEntry* (p. 24)

6.8 Unlocking the Door

FS.UserEntry.UnlockDoorOK

ScGainInitial.Suc.UserCard	*ScGainRepeat.Suc.UserCard*
ScGainInitial.Suc.UserIn	*ScGainRepeat.Suc.UserIn*
ScGainInitial.Suc.Audit	*ScGainRepeat.Suc.Audit*

The door will only be unlocked once the user has removed their token. This helps remind the user to take their token with them.

UnlockDoorOK

$$
\begin{array}{l}
UserEntryContext \\
\\
\Xi UserToken \\
\\
UnlockDoor \\
AddSuccessfulEntryToStats \\
AddElementsToLog \\
\hline
status = waitingRemoveTokenSuccess \\
userTokenPresence = absent \\
\\
currentDisplay' = doorUnlocked \\
status' = quiescent
\end{array}
$$

▷ See: *UserEntryContext* (p. 44), *UserToken* (p. 23), *UnlockDoor* (p. 38), *AddSuccessfulEntryToStats* (p. 37), *AddElementsToLog* (p. 33), *absent* (p. 8), *doorUnlocked* (p. 15), *quiescent* (p. 24)

FS.UserEntry.WaitingTokenRemoval

The system will wait indefinitely for a token to be removed, however the system will deny entry to a user who takes too long to extract their token.

WaitingTokenRemoval

$$
\begin{array}{l}
\Xi IDStation \\
RealWorldChanges \\
\\
\Xi TISControlledRealWorld \\
\hline
status \in \{waitingRemoveTokenSuccess, waitingRemoveTokenFail\} \\
status = waitingRemoveTokenSuccess \Rightarrow currentTime \leq tokenRemovalTimeout \\
userTokenPresence = present
\end{array}
$$

▷ See: *IDStation* (p. 25), *RealWorldChanges* (p. 27), *TISControlledRealWorld* (p. 17), *waitingRemoveTokenFail* (p. 24), *present* (p. 8)

FS.UserEntry.TokenRemovalTimeout

If the user waits too long to remove their token then this is logged and the system continues to wait for the token to be removed but will no longer allow access to the enclave.

TokenRemovalTimeout

UserEntryContext

Ξ*DoorLatchAlarm*
Ξ*UserToken*
Ξ*Stats*

AddElementsToLog

---

status $=$ *waitingRemoveTokenSuccess*
currentTime $>$ *tokenRemovalTimeout*
userTokenPresence $=$ *present*

status$'$ $=$ *waitingRemoveTokenFail*
currentDisplay$'$ $=$ *removeToken*

▷ See: *UserEntryContext* (p. 44), *DoorLatchAlarm* (p. 22), *UserToken* (p. 23), *Stats* (p. 21), *AddElementsToLog* (p. 33), *present* (p. 8), *waitingRemoveTokenFail* (p. 24)

TISUnlockDoor $\widehat{=}$ *UnlockDoorOK*
$\lor$ [*WaitingTokenRemoval* | *status* $=$ *waitingRemoveTokenSuccess*]
$\lor$ *TokenRemovalTimeout*

▷ See: *UnlockDoorOK* (p. 55), *WaitingTokenRemoval* (p. 55), *TokenRemovalTimeout* (p. 55)

6.9 Terminating a failed access

FS.UserEntry.FailedAccessTokenRemoved

ScGainInitial.Suc.Audit *ScProhibitInitial.Suc.Audit*
ScProhibitInitial.Suc.UserCard

If an access attempt has failed the system waits for the token to be removed before a new user entry operation can commence. Once the token has been removed a new user entry may start.

The operations in the enclave are not blocked on the presence of a failed user token in the token reader.

FailedAccessTokenRemoved

UserEntryContext

Ξ*UserToken*
Ξ*DoorLatchAlarm*

AddFailedEntryToStats
AddElementsToLog

---

status $=$ *waitingRemoveTokenFail*
userTokenPresence $=$ *absent*

currentDisplay$'$ $=$ *welcome*
status$'$ $=$ *quiescent*

▷ See: *UserEntryContext* (p. 44), *UserToken* (p. 23), *DoorLatchAlarm* (p. 22), *AddFailedEntryToStats* (p. 37), *AddElementsToLog* (p. 33), *waitingRemoveTokenFail* (p. 24), *absent* (p. 8), *welcome* (p. 15), *quiescent* (p. 24)

TISCompleteFailedAccess $\widehat{=}$ *FailedAccessTokenRemoved*
$\lor$ [*WaitingTokenRemoval* | *status* $=$ *waitingRemoveTokenFail*]

▷ See: *FailedAccessTokenRemoved* (p. 56), *WaitingTokenRemoval* (p. 55), *waitingRemoveTokenFail* (p. 24)

6.10 The Complete User Entry

FS.UserEntry.TISUserEntryOp
FIA_UAU.7.1

The complete authentication process, triggered by TIS reading a User Token, involves validating the user Token, reading and validating the fingerprint, writing an authorisation certificate to the user token, waiting for the user to remove the token, opening the door to the enclave and in the case of a failure waiting for the system to be in a state where it can admit another user.

$$TISUserEntryOp \mathrel{\widehat=} TISReadUserToken \lor TISValidateUserToken \lor TISReadFinger \lor TISValidateFinger \lor TISWriteUserToken \lor TISValidateEntry \lor TISUnlockDoor \lor TISCompleteFailedAccess$$

▷ See: *TISReadUserToken* (p. 46), *TISValidateUserToken* (p. 49), *TISReadFinger* (p. 50), *TISValidateFinger* (p. 52), *TISWriteUserToken* (p. 53), *TISValidateEntry* (p. 54), *TISUnlockDoor* (p. 56), *TISCompleteFailedAccess* (p. 56)

7 OPERATIONS WITHIN THE ENCLAVE

A number of interactions with TIS may occur within the Enclave. These interactions leave some of the *IDStation* state unchanged.

EnclaveContext

$\Delta IDStation$
RealWorldChanges

$\Xi TISControlledRealWorld$

$\Xi UserToken$
$\Xi AdminToken$
$\Xi Finger$
$\Xi Stats$

$tokenRemovalTimeout' = tokenRemovalTimeout$

▷ See: *IDStation* (p. 25), *RealWorldChanges* (p. 27), *TISControlledRealWorld* (p. 17), *UserToken* (p. 23), *AdminToken* (p. 23), *Finger* (p. 23), *Stats* (p. 21)

▷ The following state components may change *KeyStore*, *Floppy*, *Config*, *Admin*, *Keyboard*, *DoorLatchAlarm*, *Internal* and *AuditLog*.

▷ The components of the real world controlled by TIS remain unchanged.

The operations that may occur within the enclave include administrator operations and the ID station enrolment. These are described in this section.

7.1 Enrolment of an ID Station

FS.Enclave.TISEnrolOp

Before TIS can be used it must be enrolled.

We assume that the initial enrolment is the only possible enrolment activity.

Enrolment is a multi-phase activity, the state transistions for an enrolment are given in Figure 7.1. Before enrolment the system is in state *notEnrolled* and, on successful completion, it enters the *quiescent* state.

The context for all enrolment operations is given below.

EnrolContext

EnclaveContext

$\Xi Keyboard$
$\Xi Admin$
$\Xi DoorLatchAlarm$
$\Xi Config$
$\Xi Floppy$

▷ See: *EnclaveContext* (p. 58), *Keyboard* (p. 24), *Admin* (p. 22), *DoorLatchAlarm* (p. 22), *Config* (p. 19), *Floppy* (p. 23)

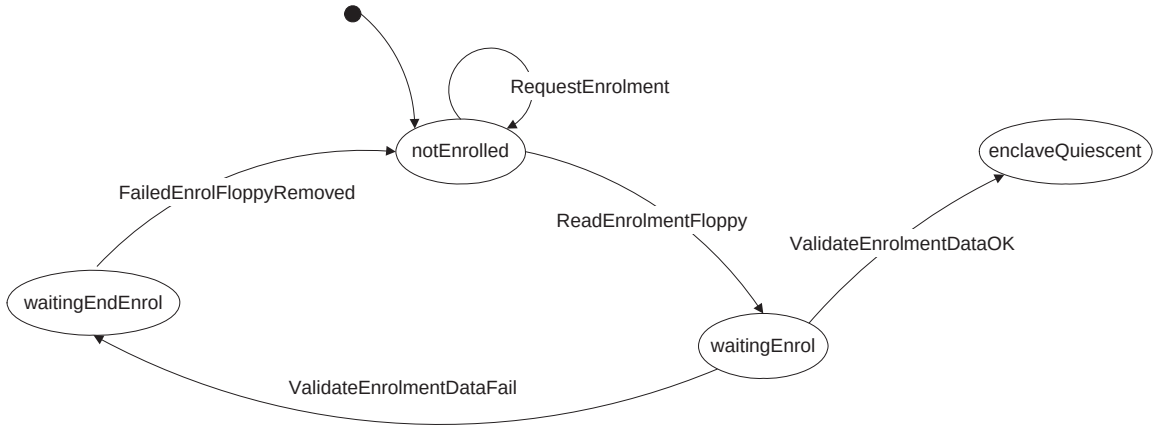

Figure 7.1: Enrolment state transitions

▷ The following state components may change *KeyStore*, *Internal* and *AuditLog*.

7.1.1 Requesting Enrolment

FS.Enclave.RequestEnrolment

The ID station will request enrolment while there is no Floppy present. This will occur until a successful enrolment is achieved.

RequestEnrolment

$$
\begin{array}{l}
EnrolContext \\
\\
\Xi KeyStore \\
\Xi AuditLog \\
\Xi Internal \\
\hline
enclaveStatus = notEnrolled \\
floppyPresence = absent \\
currentScreen'.screenMsg = insertEnrolmentData \\
currentDisplay' = blank
\end{array}
$$

▷ See: *EnrolContext* (p. 58), *KeyStore* (p. 21), *AuditLog* (p. 20), *Internal* (p. 24), *notEnrolled* (p. 24), *absent* (p. 8), *blank* (p. 15)

FS.Enclave.ReadEnrolmentFloppy

ScStart.Ass.Data *ScStart.Con.NoInterleave*

If a floppy is present then TIS goes on to validate the contents. Nothing is written to the log at this stage as log entries will be made on successful or failed enrolment.

ReadEnrolmentFloppy

$$
\begin{array}{l}
\mathit{EnrolContext} \\
\\
\Xi \mathit{KeyStore} \\
\hline
\mathit{enclaveStatus} = \mathit{notEnrolled} \\
\mathit{floppyPresence} = \mathit{present} \\
\\
\mathit{currentScreen}'.\mathit{screenMsg} = \mathit{validatingEnrolmentData} \\
\\
\mathit{enclaveStatus}' = \mathit{waitingEnrol} \\
\mathit{status}' = \mathit{status} \\
\mathit{currentDisplay}' = \mathit{blank}
\end{array}
$$

▷ See: *EnrolContext* (p. 58), *KeyStore* (p. 21), *notEnrolled* (p. 24), *present* (p. 8), *validatingEnrolmentData* (p. 16), *waitingEnrol* (p. 24), *blank* (p. 15)

$$\mathit{ReadEnrolmentData} \mathrel{\widehat=} \mathit{ReadEnrolmentFloppy} \lor \mathit{RequestEnrolment}$$

▷ See: *ReadEnrolmentFloppy* (p. 59), *RequestEnrolment* (p. 59)

7.1.2 Validating Enrolment data from Floppy

For the enrolment data to be acceptable the data on the floppy must be valid enrolment data with the ID Station certificate containing this ID station's public key.

EnrolmentDataOK

$$
\begin{array}{l}
\mathit{Floppy} \\
\mathit{KeyStore} \\
\hline
\mathit{currentFloppy} \in \operatorname{ran} \mathit{enrolmentFile} \\
(\exists\, \mathit{ValidEnrol} \bullet \theta \mathit{ValidEnrol} = \mathit{enrolmentFile}^{\sim} \mathit{currentFloppy})
\end{array}
$$

▷ See: *Floppy* (p. 23), *KeyStore* (p. 21), *enrolmentFile* (p. 16), *ValidEnrol* (p. 14)

FS.Enclave.ValidateEnrolmentDataOK	
ScStart.Suc.Running	*FMT_MSA.2.1*
ScStart.Suc.Audit	*FMT_MTD.3.1*

If the data on the floppy is acceptable to be used for enrolment then the Key store is updated. From this point the system is available for use both by users entering the enclave and by administrators.

ValidateEnrolmentDataOK

$$
\begin{array}{l}
\mathit{EnrolContext} \\
\\
\mathit{UpdateKeyStoreFromFloppy} \\
\mathit{AddElementsToLog} \\
\hline
\mathit{enclaveStatus} = \mathit{waitingEnrol} \\
\\
\mathit{EnrolmentDataOK} \\
\\
\mathit{currentScreen}'.\mathit{screenMsg} = \mathit{welcomeAdmin} \\
\\
\mathit{enclaveStatus}' = \mathit{enclaveQuiescent} \\
\mathit{status}' = \mathit{quiescent} \\
\mathit{currentDisplay}' = \mathit{welcome}
\end{array}
$$

▷ See: *EnrolContext* (p. 58), *UpdateKeyStoreFromFloppy* (p. 40), *AddElementsToLog* (p. 33), *waitingEnrol* (p. 24), *EnrolmentDataOK* (p. 60), *welcomeAdmin* (p. 16), *quiescent* (p. 24), *welcome* (p. 15)

FS.Enclave.ValidateEnrolmentDataFail
ScStart.Fail.ReadFloppy

If the enrolment fails then TIS waits for the floppy to be removed before prompting for new enrolment data.

ValidateEnrolmentDataFail

$$
\begin{array}{l}
EnrolContext \\
\Xi KeyStore \\
AddElementsToLog \\
\hline
enclaveStatus = waitingEnrol \\
\neg\ EnrolmentDataOK \\
currentScreen'.screenMsg = enrolmentFailed \\
enclaveStatus' = waitingEndEnrol \\
status' = status \\
currentDisplay' = blank
\end{array}
$$

▷ See: *EnrolContext* (p. 58), *KeyStore* (p. 21), *AddElementsToLog* (p. 33), *waitingEnrol* (p. 24), *EnrolmentDataOK* (p. 60), *enrolmentFailed* (p. 16), *waitingEndEnrol* (p. 24), *blank* (p. 15)

$$ValidateEnrolmentData \mathrel{\widehat=} ValidateEnrolmentDataOK \lor ValidateEnrolmentDataFail$$

▷ See: *ValidateEnrolmentDataOK* (p. 60), *ValidateEnrolmentDataFail* (p. 61)

7.1.3 Completing a failed Enrolment

A failed enrolment will only terminate once the floppy has been removed, otherwise the system would repeatedly try to validate the same floppy.

FS.Enclave.FailedEnrolFloppyRemoved

Once the floppy has been removed the administrator is prompted for enrolment data again. We do not log the removal of the floppy in the audit log.

FailedEnrolFloppyRemoved

$$
\begin{array}{l}
EnrolContext \\
\Xi KeyStore \\
\hline
enclaveStatus = waitingEndEnrol \\
floppyPresence = absent \\
currentScreen'.screenMsg = insertEnrolmentData \\
enclaveStatus' = notEnrolled \\
status' = status \\
currentDisplay' = blank
\end{array}
$$

▷ See: *EnrolContext* (p. 58), *KeyStore* (p. 21), *waitingEndEnrol* (p. 24), *absent* (p. 8), *notEnrolled* (p. 24), *blank* (p. 15)

FS.Enclave.WaitingFloppyRemoval

TIS will wait indefinately for the floppy to be removed after an unsuccessful enrolment, this is because enrolment is triggered by the presence of the floppy alone.

WaitingFloppyRemoval

$$\begin{array}{l} EnclaveContext \\ \\ \Xi IDStation \\ \hline enclaveStatus = waitingEndEnrol \\ floppyPresence = present \end{array}$$

▷ See: *EnclaveContext* (p. 58), *IDStation* (p. 25), *waitingEndEnrol* (p. 24), *present* (p. 8)

$$CompleteFailedEnrolment \mathrel{\widehat=} FailedEnrolFloppyRemoved \lor WaitingFloppyRemoval$$

▷ See: *FailedEnrolFloppyRemoved* (p. 61), *WaitingFloppyRemoval* (p. 62)

7.1.4 The Complete Enrolment

The complete enrolment process involves reading the enrolment data, validating it and, in the case of a failure waiting for the system to be in a state where it can try another enrolment.

$$\begin{array}{l} TISEnrolOp \mathrel{\widehat=} ReadEnrolmentData \lor ValidateEnrolmentData \\ \qquad\qquad \lor CompleteFailedEnrolment \end{array}$$

▷ See: *ReadEnrolmentData* (p. 60), *ValidateEnrolmentData* (p. 61), *CompleteFailedEnrolment* (p. 62)

7.2 Administrator Token Tear

The action of removing the administrator Token will result in the administrator being logged out of the system.

This may happen at any point once a token has been inserted into the reader. As soon as the adminitrator's token is torn this action will be logged. The screen message will be reset if the system is not busy with processing a user entry.

AdminTokenTear

$$\begin{array}{l} EnclaveContext \\ \\ \Xi Config \\ \Xi Floppy \\ \Xi Keyboard \\ \Xi DoorLatchAlarm \\ \Xi KeyStore \\ ResetScreenMessage \\ \hline adminTokenPresence = absent \\ \\ status' = status \\ currentDisplay' = currentDisplay \\ \\ enclaveStatus' = enclaveQuiescent \end{array}$$

▷ See: *EnclaveContext* (p. 58), *Config* (p. 19), *Floppy* (p. 23), *Keyboard* (p. 24), *DoorLatchAlarm* (p. 22), *KeyStore* (p. 21), *ResetScreenMessage* (p. 44), *absent* (p. 8)

If the admin token is torn while the system is processing an activity within the enclave then the activity will be stopped.

BadAdminTokenTear

$$\begin{array}{l} \textit{AdminTokenTear} \\ \textit{AddElementsToLog} \\ \hline \textit{enclaveStatus} \in \{\textit{gotAdminToken}, \textit{waitingStartAdminOp}, \textit{waitingFinishAdminOp}\} \end{array}$$

▷ See: *AdminTokenTear* (p. 62), *AddElementsToLog* (p. 33), *waitingStartAdminOp* (p. 24), *waitingFinishAdminOp* (p. 24)

FS.Enclave.BadAdminLogout

ScLogOff.Ass.LoggedOn *ScLogOff.Suc.Audit*
ScLogOff.Suc.LoggedOff

If the administrator is performing an operation when the token is torn then the administrator will be logged off.

BadAdminLogout

$$\begin{array}{l} \textit{BadAdminTokenTear} \\ \textit{AdminLogout} \\ \hline \textit{enclaveStatus} \in \{\textit{waitingStartAdminOp}, \textit{waitingFinishAdminOp}\} \end{array}$$

▷ See: *BadAdminTokenTear* (p. 63), *AdminLogout* (p. 41), *waitingStartAdminOp* (p. 24), *waitingFinishAdminOp* (p. 24)

FS.Enclave.LoginAborted

If the token is torn during the log on validation process then there is no need to log off the administrator.

LoginAborted

$$\begin{array}{l} \textit{BadAdminTokenTear} \\ \Xi \textit{Admin} \\ \hline \textit{enclaveStatus} = \textit{gotAdminToken} \end{array}$$

▷ See: *BadAdminTokenTear* (p. 63), *Admin* (p. 22)

7.3 Administrator Login

An Administrator logs into TIS by inserting a valid token into the *adminToken* reader. The authorisation certificate is verified and the user is logged in with the privileges indicated on the card.

Once the administrator is successfully logged into TIS, the system records that there is a role present. The process of logging on is given by the state transition diagram in Figure 7.2

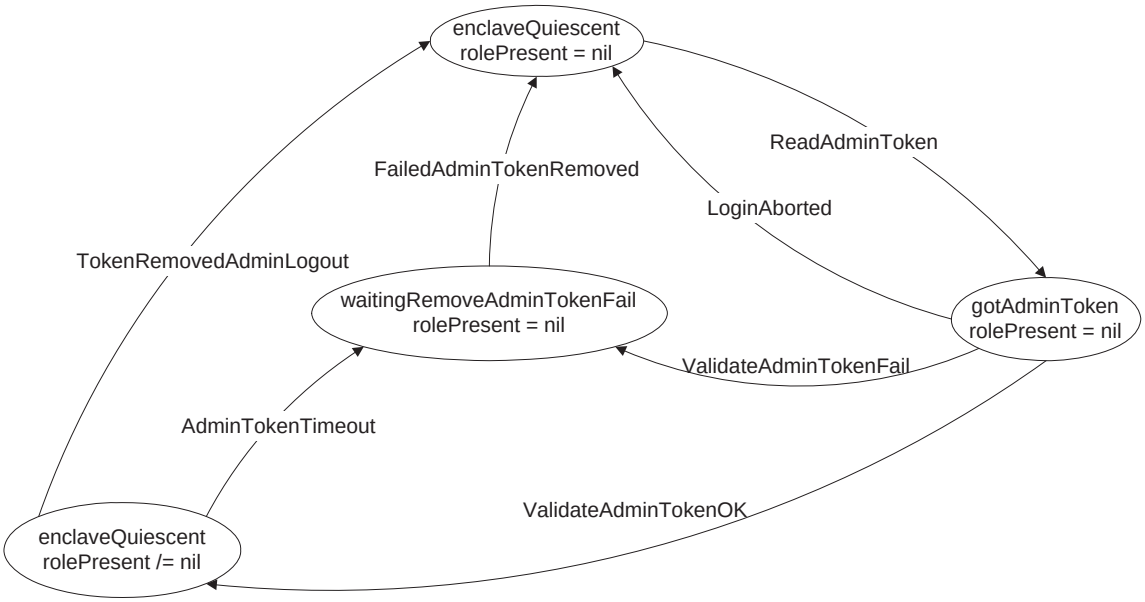

Figure 7.2: Administrator logon/logoff state transitions

The context for administrator login is given below.

LoginContext

$$
\begin{array}{l}
\textit{EnclaveContext} \\
\\
\Xi \textit{Keyboard} \\
\Xi \textit{KeyStore} \\
\Xi \textit{DoorLatchAlarm} \\
\Xi \textit{Config} \\
\Xi \textit{Floppy} \\
\hline
\textit{status}' = \textit{status} \\
\textit{currentDisplay}' = \textit{currentDisplay}
\end{array}
$$

▷ See: *EnclaveContext* (p. 58), *Keyboard* (p. 24), *KeyStore* (p. 21), *DoorLatchAlarm* (p. 22), *Config* (p. 19), *Floppy* (p. 23)

▷ The following state components may change *Admin*, *Internal* and *AuditLog*.

7.3.1 Read Administrator Token

FS.Enclave.ReadAdminToken	
ScLogOn.Ass.Quiescent	*FIA_UID.2.1*
ScLogOn.Suc.Audit	*FMT_SMR.3.1*
ScLogOn.Con.NoInterleave	

When the admin token is read the action is audited and the internal status changes. No other aspects of the system are modified.

An administrator can only log on when there is no user entry activity in progress or TIS is waiting for a failed user token to be removed from the token reader outside of the enclave.

ReadAdminToken

$$
\begin{array}{l}
LoginContext \\
\\
\Xi Admin \\
\\
AddElementsToLog \\
\hline
status \in \{ quiescent, waitingRemoveTokenFail \} \\
\\
enclaveStatus = enclaveQuiescent \\
rolePresent = nil \\
adminTokenPresence = present \\
\\
enclaveStatus' = gotAdminToken \\
currentScreen' = currentScreen
\end{array}
$$

▷ See: *LoginContext* (p. 64), *Admin* (p. 22), *AddElementsToLog* (p. 33), *quiescent* (p. 24), *waitingRemoveTokenFail* (p. 24), *nil* (p. 8), *present* (p. 8)

The operation to read the token is as follows:

$$TISReadAdminToken \mathrel{\widehat=} ReadAdminToken$$

▷ See: *ReadAdminToken* (p. 65)

7.3.2 Validate Administrator Token

An administrator's token is considered valid if it cotains a current authorisation certificate that correctly cross references to the token ID and the ID certificate and both these certificates can be validated using the keys held in the *KeyStore*. Additionally the privileges assigned to the user within the authorisation certificate must indicate that the user is actually an administrator.

AdminTokenOK

$$
\begin{array}{l}
AdminToken \\
KeyStore \\
currentTime : TIME \\
\hline
currentAdminToken \in \operatorname{ran} goodT \\
\\
\exists\, TokenWithValidAuth \bullet \\
\quad (goodT(\theta TokenWithValidAuth) = currentAdminToken \\
\quad \land\ (\exists\, IDCert \bullet \theta IDCert = idCert \land CertOK) \\
\quad \land\ (\exists\, AuthCert \bullet \theta AuthCert = the\ authCert \land AuthCertOK) \\
\quad \land\ (the\ authCert).role \in ADMINPRIVILEGE \\
\quad \land\ currentTime \in (the\ authCert).validityPeriod)
\end{array}
$$

▷ See: *AdminToken* (p. 23), *KeyStore* (p. 21), *TIME* (p. 8), *goodT* (p. 15), *TokenWithValidAuth* (p. 13), *IDCert* (p. 11), *CertOK* (p. 39), *AuthCert* (p. 12), *the* (p. 8), *AuthCertOK* (p. 39), *ADMINPRIVILEGE* (p. 22)

▷ Only the *AuthCert* and *IDCert* are checked at this point. The remaining certificates were checked on entry to the enclave.

▷ The Token must indicate that the user has an administrator privilege.

FS.Enclave.ValidateAdminTokenOK

ScLogOn.Ass.ValidAdmin	*FDP_ACF.1.2*
ScLogOn.Suc.LogOn	*FDP_ACF.1.3*
ScLogOn.Suc.Audit	*FDP_ACF.1.4*
SFP.DAC	*FIA_USB.1.1*
FCO_NRO.2.1	*FMT_MSA.1.1*
FCO_NRO.2.2	*FMT_MTD.1.1*
FCO_NRO.2.3	*FMT_SAE.1.1*
FDP_ACC.1.1	*FMT_SMR.2.1*
FDP_ACF.1.1	*FMT_SMR.2.2*

If the token can be validated then the administrator is logged onto TIS.

ValidateAdminTokenOK

$$
\begin{array}{l}
LoginContext \\
AdminLogon \\
AddElementsToLog \\
\hline
enclaveStatus = gotAdminToken \\
adminTokenPresence = present \\
AdminTokenOK \\
currentScreen'.screenMsg = requestAdminOp \\
enclaveStatus' = enclaveQuiescent
\end{array}
$$

▷ See: *LoginContext* (p. 64), *AdminLogon* (p. 41), *AddElementsToLog* (p. 33), *present* (p. 8), *AdminTokenOK* (p. 65)

FS.Enclave.ValidateAdminTokenFail

ScLogOn.Fail.ReadCard

If the token can not be validated then TIS waits for it to be removed.

ValidateAdminTokenFail

$$
\begin{array}{l}
LoginContext \\
\Xi Admin \\
AddElementsToLog \\
\hline
enclaveStatus = gotAdminToken \\
adminTokenPresence = present \\
\neg\ AdminTokenOK \\
currentScreen'.screenMsg = removeAdminToken \\
enclaveStatus' = waitingRemoveAdminTokenFail
\end{array}
$$

▷ See: *LoginContext* (p. 64), *Admin* (p. 22), *AddElementsToLog* (p. 33), *present* (p. 8), *AdminTokenOK* (p. 65), *removeAdminToken* (p. 16), *waitingRemoveAdminTokenFail* (p. 24)

$$
TISValidateAdminToken \mathrel{\widehat=} ValidateAdminTokenOK \lor ValidateAdminTokenFail \lor LoginAborted
$$

▷ See: *ValidateAdminTokenOK* (p. 66), *ValidateAdminTokenFail* (p. 66), *LoginAborted* (p. 63)

7.3.3 Complete Failed Administrator Logon

If an administrator token has failed to be accepted by TIS then no further actions can take place in the enclave until it has been removed.

FS.Enclave.FailedAdminTokenRemoved

The administrator token may be removed at any point during a user entry, hence the context for this activity does not place restrictions on the value of *status*.

When the admin token is removed TIS returns to a state ready to accept another administrator logon.

FailedAdminTokenRemoved

$$
\begin{array}{l}
LoginContext \\
\Xi Admin \\
AddElementsToLog \\
\hline
enclaveStatus = waitingRemoveAdminTokenFail \\
adminTokenPresence = absent \\
currentScreen'.screenMsg = welcomeAdmin \\
enclaveStatus' = enclaveQuiescent \\
currentDisplay' = currentDisplay
\end{array}
$$

▷ See: *LoginContext* (p. 64), *Admin* (p. 22), *AddElementsToLog* (p. 33), *waitingRemoveAdminTokenFail* (p. 24), *absent* (p. 8), *welcomeAdmin* (p. 16)

FS.Enclave.WaitingAdminTokenRemoval

TIS will wait indefinitely for the Admin Token to be removed after a failed attempt to logon.

WaitingAdminTokenRemoval

$$
\begin{array}{l}
EnclaveContext \\
\Xi IDStation \\
\hline
enclaveStatus = waitingRemoveAdminTokenFail \\
adminTokenPresence = present
\end{array}
$$

▷ See: *EnclaveContext* (p. 58), *IDStation* (p. 25), *waitingRemoveAdminTokenFail* (p. 24), *present* (p. 8)

$$TISCompleteFailedAdminLogon \mathrel{\widehat=} FailedAdminTokenRemoved \lor WaitingAdminTokenRemoval$$

▷ See: *FailedAdminTokenRemoved* (p. 67), *WaitingAdminTokenRemoval* (p. 67)

7.3.4 The Complete Administrator Logon

FS.Enclave.TISAdminLogin

The complete administrator logon process, from the point that the system has detected the presence of a token in the administrator reader, involves validating the administrator token and, in the case of a failure waiting for the system to be in a state where it can try another logon.

$$TISAdminLogon \mathrel{\widehat=} TISReadAdminToken \lor TISValidateAdminToken \lor TISCompleteFailedAdminLogon$$

▷ See: *TISReadAdminToken* (p. 65), *TISValidateAdminToken* (p. 66), *TISCompleteFailedAdminLogon* (p. 67)

7.4 Administrator Logout

Administrator logout can be achieved in two ways, either the administrator removes their token from TIS, or the Authorisation certificate on the token expires, causing the system to automatically log off the administrator.

FS.Enclave.AdminLogout

ScLogOff.Ass.LoggedOn *ScLogOff.Suc.Audit*
ScLogOff.Suc.LoggedOff

If TIS is not performing an administrator operation then the token may be removed to log out the administrator.

TokenRemovedAdminLogout

$$AdminTokenTear$$
$$AdminLogout$$
$$AddElementsToLog$$

$$enclaveStatus = enclaveQuiescent$$
$$rolePresent \neq nil$$

▷ See: *AdminTokenTear* (p. 62), *AdminLogout* (p. 41), *AddElementsToLog* (p. 33), *nil* (p. 8)

FS.Enclave.AdminTokenTimeout

The TIS will automatically logout an administrator whose token expires. This occurs if the validity period on the Authorisation certificate expires.

AdminTokenTimeout

$$LoginContext$$
$$AdminLogout$$
$$AddElementsToLog$$
$$ResetScreenMessage$$

$$enclaveStatus = enclaveQuiescent$$
$$adminTokenPresence = present$$
$$rolePresent \neq nil$$
$$\neg\ AdminTokenOK$$
$$enclaveStatus' = waitingRemoveAdminTokenFail$$

▷ See: *LoginContext* (p. 64), *AdminLogout* (p. 41), *AddElementsToLog* (p. 33), *ResetScreenMessage* (p. 44), *present* (p. 8), *nil* (p. 8), *AdminTokenOK* (p. 65), *waitingRemoveAdminTokenFail* (p. 24)

FS.Enclave.TISCompleteTimeoutAdminLogout

If the administrator's token expires then it must be removed before further activities can take place at the TIS console. This behaviour is identical to the behaviour when the system waits for a the administrator to remove their token following a failed logon.

$$TISCompleteTimeoutAdminLogout \mathrel{\widehat=} TISCompleteFailedAdminLogon$$

▷ See: *TISCompleteFailedAdminLogon* (p. 67)

7.4.1 Complete Administrator Logout

FS.Enclave.TISAdminLogout

THe complete administrator logout process, from the point that it decides to log out an administrator to the point that it is in a state where it can try another logon is givien below.

$$TISAdminLogout \mathrel{\widehat=} TokenRemovedAdminLogout \lor AdminTokenTimeout \lor TISCompleteTimeoutAdminLogout$$

▷ See: *TokenRemovedAdminLogout* (p. 68), *AdminTokenTimeout* (p. 68), *TISCompleteTimeoutAdminLogout* (p. 69)

7.5 Administrator Operations

An administrator operation can take place as long as an administrator is present. The operation is started by receiving a valid request to perform an operation from the keyboard. TIS will ensure that the requested operation is one compatible with the current role present.

Once the operation is started the behaviour depends on the type of operation. Operations are either short, and can be implemented in one phase or they are multi-phase operations.

shutdown and *overrideLock* are short operations, while *archiveLog* and *updateCofigData* are multi phase operations.

The state transition diagram for administrator operations is given in Figure 7.3

All administrator operations have a common context, in which the *AdminToken* does not change. An administrator can only perform an operation when there is no user entry activity in progress or TIS is waiting for a failed user token to be removed from the token reader outside of the enclave.

AdminOpContext

$$\begin{array}{l} EnclaveContext \\ \\ \Xi Keyboard \\ \Xi KeyStore \end{array}$$

▷ See: *EnclaveContext* (p. 58), *Keyboard* (p. 24), *KeyStore* (p. 21)

▷ The following state components may change *Floppy*, *Config*, *Admin*, *DoorLatchAlarm*, *Internal* and *AuditLog*.

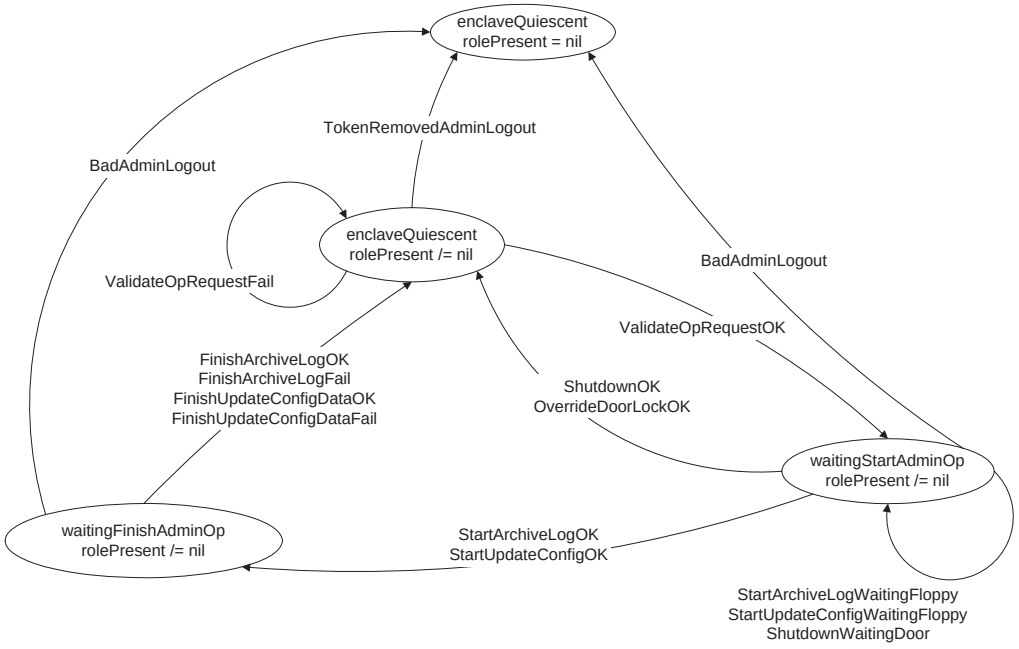

Figure 7.3: Administrator operation state transitions

Once an operation has been started its context is given by:

AdminOpStartedContext

AdminOpContext

$enclaveStatus = waitingStartAdminOp$
$adminTokenPresence = present$

$status' = status$

▷ See: *AdminOpContext* (p. 69), *waitingStartAdminOp* (p. 24), *present* (p. 8)

▷ The *adminToken* will be present, its removal is erroneous.

▷ The system has a record of the name of the current operation.

Some operations are multi-phase, the context for completing a multi-phase operation is given by:

AdminOpFinishContext

AdminOpContext

AdminFinishOp

$enclaveStatus = waitingFinishAdminOp$
$adminTokenPresence = present$

$status' = status$
$currentDisplay' = currentDisplay$

$enclaveStatus' = enclaveQuiescent$

▷ See: *AdminOpContext* (p. 69), *AdminFinishOp* (p. 42), *waitingFinishAdminOp* (p. 24), *present* (p. 8)

▷ The *adminToken* will be present, its removal is erroneous.

▷ The enclaveStatus value implies that TIS has a record of the name of the current operation from the *IDStation* invariant.

7.6 Starting Operations

All administrator operations are initiated in the same way. This involves validating the latest keyboard input and determining whether it is a valid operation request.

TIS only attempts to start an operation if there is an administrator present and there is no current activity in the enclave. An administrator can only start an operation when there is no user entry activity in progress or TIS is waiting for a failed user token to be removed from the token reader outside of the enclave.

StartOpContext

$$
\begin{array}{l}
EnclaveContext \\
\\
\Xi DoorLatchAlarm \\
\Xi Keyboard \\
\Xi Config \\
\Xi Floppy \\
\Xi KeyStore \\
\hline
enclaveStatus = enclaveQuiescent \\
adminTokenPresence = present \\
rolePresent \neq nil \\
status \in \{ quiescent, waitingRemoveTokenFail \} \\
\\
status' = status \\
currentDisplay' = currentDisplay
\end{array}
$$

▷ See: *EnclaveContext* (p. 58), *DoorLatchAlarm* (p. 22), *Keyboard* (p. 24), *Config* (p. 19), *Floppy* (p. 23), *KeyStore* (p. 21), *present* (p. 8), *nil* (p. 8), *quiescent* (p. 24), *waitingRemoveTokenFail* (p. 24)

▷ The following state components may change *Admin*, *Internal* and *AuditLog*.

7.6.1 Validating an Operation Request

FS.Enclave.ValidateOpRequestOK	
ScShutdown.Suc.Audit	*FDP_ACF.1.4*
ScConfig.Suc.Audit	*FIA_USB.1.1*
ScUnlock.Suc.Audit	*FMT_MOF.1.1*
SFP.DAC	*FMT_MSA.1.1*
FDP_ACC.1.1	*FMT_MTD.1.1*
FDP_ACF.1.1	*FMT_SMR.2.1*
FDP_ACF.1.2	*FMT_SAE.1.1*
FDP_ACF.1.3	

Once the data from the keyboard has been read this must be validated to ensure it corresponds to a valid operation.

ValidateOpRequestOK

$$
\begin{array}{l}
StartOpContext \\
AdminStartOp \\
AddElementsToLog \\
\hline
keyedDataPresence = present \\
currentKeyedData \in keyedOps(\!|availableOps|\!) \\
currentScreen'.screenMsg = doingOp \\
enclaveStatus' = waitingStartAdminOp
\end{array}
$$

▷ See: *StartOpContext* (p. 71), *AdminStartOp* (p. 42), *AddElementsToLog* (p. 33), *present* (p. 8), *keyedOps* (p. 16), *doingOp* (p. 16), *waitingStartAdminOp* (p. 24)

FS.Enclave.ValidateOpRequestFail

If the data from the keyboard doesn't correspond to an operation that can be performed at present then the operation is not started and the attempt to start an illegal operation is logged.

ValidateOpRequestFail

$$
\begin{array}{l}
StartOpContext \\
\Xi Admin \\
AddElementsToLog \\
\hline
keyedDataPresence = present \\
currentKeyedData \notin keyedOps(\!|availableOps|\!) \\
currentScreen'.screenMsg = invalidRequest \\
enclaveStatus' = enclaveStatus
\end{array}
$$

▷ See: *StartOpContext* (p. 71), *Admin* (p. 22), *AddElementsToLog* (p. 33), *present* (p. 8), *keyedOps* (p. 16), *invalidRequest* (p. 16)

FS.Enclave.NoOpRequest

If there is no data at the keyboard then TIS waits for user interaction.

NoOpRequest

$$
\begin{array}{l}
StartOpContext \\
\Xi IDStation \\
\hline
keyedDataPresence = absent
\end{array}
$$

▷ See: *StartOpContext* (p. 71), *IDStation* (p. 25), *absent* (p. 8)

$$ValidateOpRequest \mathrel{\widehat=} ValidateOpRequestOK \lor ValidateOpRequestFail \lor NoOpRequest$$

▷ See: *ValidateOpRequestOK* (p. 71), *ValidateOpRequestFail* (p. 72), *NoOpRequest* (p. 72)

7.6.2 Complete Operation Start

FS.Enclave.TISStartAdminOp

The process of starting an administrator operation involves exactly the validation of an operation request.

$$TISStartAdminOp \mathrel{\widehat=} ValidateOpRequest$$

▷ See: *ValidateOpRequest* (p. 72)

7.7 Archiving the Log

When the log is archived it is copied to floppy and the internally held log is truncated.

The internally held log can only be truncated if the write to floppy succeeds.

To check that the archive succeeded the floppy is read back and the data compared with that held by the system.

This is a two phase operation, during the first phase the log is written to floppy, during the second phase the data on the floppy is validated.

7.7.1 Writing the archive Log

FS.Enclave.StartArchiveLogOK

ScAudit.Ass.LoggedOn *ScAudit.Con.NoInterleave*

The first phase of this operation is to write the archive log to floppy.

StartArchiveLogOK

$$\begin{array}{l} AdminOpStartedContext \\ \\ \Xi Config \\ \Xi Admin \\ \Xi DoorLatchAlarm \\ \hline the\ currentAdminOp = archiveLog \\ floppyPresence = present \\ \\ floppyPresence' = floppyPresence \\ currentFloppy' = currentFloppy \\ \\ currentScreen'.screenMsg = doingOp \\ currentDisplay' = currentDisplay \\ \\ enclaveStatus' = waitingFinishAdminOp \\ (\exists\, archive : \mathbb{F}\, Audit \bullet ArchiveLog \land writtenFloppy' = auditFile\, archive) \end{array}$$

▷ See: *AdminOpStartedContext* (p. 70), *Config* (p. 19), *Admin* (p. 22), *DoorLatchAlarm* (p. 22), *the* (p. 8), *archiveLog* (p. 22), *present* (p. 8), *doingOp* (p. 16), *waitingFinishAdminOp* (p. 24), *ArchiveLog* (p. 34)

FS.Enclave.StartArchiveLogWaitingFloppy

We wait indefinitely for a floppy to be present.

StartArchiveLogWaitingFloppy

$$
\begin{array}{l}
AdminOpStartedContext \\
\\
\Xi Config \\
\Xi Admin \\
\Xi DoorLatchAlarm \\
\Xi Floppy \\
\hline
the\ currentAdminOp = archiveLog \\
floppyPresence = absent \\
\\
currentScreen'.screenMsg = insertBlankFloppy \\
currentDisplay' = currentDisplay \\
\\
enclaveStatus' = enclaveStatus
\end{array}
$$

▷ See: *AdminOpStartedContext* (p. 70), *Config* (p. 19), *Admin* (p. 22), *DoorLatchAlarm* (p. 22), *Floppy* (p. 23), *the* (p. 8), *archiveLog* (p. 22), *absent* (p. 8), *insertBlankFloppy* (p. 16)

$$
\begin{array}{l}
StartArchiveLog \mathrel{\widehat=} (StartArchiveLogOK \fatsemi UpdateFloppy) \\
\qquad \lor\ StartArchiveLogWaitingFloppy \\
\qquad \lor\ [\, BadAdminLogout \mid enclaveStatus = waitingStartAdminOp \\
\qquad\qquad \land\ the\ currentAdminOp = archiveLog \,]
\end{array}
$$

▷ See: *StartArchiveLogOK* (p. 73), *UpdateFloppy* (p. 31), *StartArchiveLogWaitingFloppy* (p. 74), *BadAdminLogout* (p. 63), *waitingStartAdminOp* (p. 24), *the* (p. 8), *archiveLog* (p. 22)

7.7.2 Clearing the archive Log

FS.Enclave.FinishArchiveLogOK

ScAudit.Suc.Clear *ScAudit.Suc.Written*

The audit log is only truncated after a check has been made to ensure that the actual floppy data matches what the system believes is on the floppy.

Having cleared the log an entry will be made in the log indicating that the archive was successful.

$$ClearLogThenAddElements \mathrel{\widehat=} ClearLog \fatsemi AddElementsToLog$$

▷ See: *ClearLog* (p. 34), *AddElementsToLog* (p. 33)

FinishArchiveLogOK

$$
\begin{array}{l}
AdminOpFinishContext \\
\\
\Xi Config \\
\Xi Floppy \\
\Xi DoorLatchAlarm \\
\hline
the\ currentAdminOp = archiveLog \\
floppyPresence = present \\
\\
writtenFloppy = currentFloppy \\
\\
(\exists\, archive : \mathbb{F}\ Audit \bullet ClearLogThenAddElements \land writtenFloppy = auditFile\ archive) \\
\\
currentScreen'.screenMsg = requestAdminOp
\end{array}
$$

▷ See: *AdminOpFinishContext* (p. 70), *Config* (p. 19), *Floppy* (p. 23), *DoorLatchAlarm* (p. 22), *the* (p. 8), *archiveLog* (p. 22), *present* (p. 8), *ClearLogThenAddElements* (p. 74)

FS.Enclave.FinishArchiveLogNoFloppy
ScAudit.Fail.Write

If the administrator is impatient and removes the floppy early then the archive fails as the system cannot check that the archive was taken.

FinishArchiveLogNoFloppy

$$
\begin{array}{l}
AdminOpFinishContext \\
\\
\Xi Config \\
\Xi Floppy \\
\Xi DoorLatchAlarm \\
\\
AddElementsToLog \\
\hline
the\ currentAdminOp = archiveLog \\
floppyPresence = absent \\
\\
currentScreen'.screenMsg = archiveFailed
\end{array}
$$

▷ See: *AdminOpFinishContext* (p. 70), *Config* (p. 19), *Floppy* (p. 23), *DoorLatchAlarm* (p. 22), *AddElementsToLog* (p. 33), *the* (p. 8), *archiveLog* (p. 22), *absent* (p. 8)

FS.Enclave.FinishArchiveLogBadMatch
ScAudit.Fail.Write

If the data read back from the floppy does not match what the ID station believes should be on the floppy then the archive fails.

FinishArchiveLogBadMatch

$$
\begin{array}{l}
AdminOpFinishContext \\
\\
\Xi Config \\
\Xi Floppy \\
\Xi DoorLatchAlarm \\
\\
AddElementsToLog \\
\hline
the\ currentAdminOp = archiveLog \\
floppyPresence = present \\
\\
writtenFloppy \neq currentFloppy \\
\\
currentScreen'.screenMsg = archiveFailed
\end{array}
$$

▷ See: *AdminOpFinishContext* (p. 70), *Config* (p. 19), *Floppy* (p. 23), *DoorLatchAlarm* (p. 22), *AddElementsToLog* (p. 33), *the* (p. 8), *archiveLog* (p. 22), *present* (p. 8)

$$FinishArchiveLogFail \mathrel{\widehat=} FinishArchiveLogBadMatch \lor FinishArchiveLogNoFloppy$$

$$
\begin{array}{l}
FinishArchiveLog \mathrel{\widehat=} FinishArchiveLogOK \lor FinishArchiveLogFail \\
\qquad \lor [\, BadAdminLogout \mid enclaveStatus = waitingFinishAdminOp \\
\qquad\qquad \land\ the\ currentAdminOp = archiveLog \,]
\end{array}
$$

▷ See: *FinishArchiveLogBadMatch* (p. 75), *FinishArchiveLogNoFloppy* (p. 75), *FinishArchiveLogOK* (p. 74), *BadAdminLogout* (p. 63), *waitingFinishAdminOp* (p. 24), *the* (p. 8), *archiveLog* (p. 22)

7.7.3 The complete archive Log operation

FS.Enclave.TISArchiveLogOp

Combining the start and finish phase of this operation gives the complete operation.

$$TISArchiveLogOp \mathrel{\widehat=} StartArchiveLog \lor FinishArchiveLog$$

▷ See: *StartArchiveLog* (p. 74), *FinishArchiveLog* (p. 75)

7.8 Updating Configuration Data

The operation to update the configuration data is a two phase operation. During the first phase the configuration data is read from floppy. During the second phase the configuration data provided on the floppy is checked (currently the check is purely that the data is configuration data) and the TIS configuration data is replaced by the new data.

7.8.1 Reading Configuration Data

FS.Enclave.StartUpdateConfigDataOK

ScConfig.Ass.LoggedOn	*FMT_MSA.2.1*
ScConfig.Con.NoInterleave	*FMT_MTD.3.1*

In order to update configuration data the administrator must supply replacement configuration data on a floppy disk.

StartUpdateConfigOK

$$\begin{array}{l} AdminOpStartedContext \\ \\ \Xi Floppy \\ \Xi Config \\ \Xi Admin \\ \Xi DoorLatchAlarm \\ \hline the\ currentAdminOp = updateConfigData \\ floppyPresence = present \\ \\ currentScreen'.screenMsg = doingOp \\ currentDisplay' = currentDisplay \\ \\ enclaveStatus' = waitingFinishAdminOp \end{array}$$

▷ See: *AdminOpStartedContext* (p. 70), *Floppy* (p. 23), *Config* (p. 19), *Admin* (p. 22), *DoorLatchAlarm* (p. 22), *the* (p. 8), *updateConfigData* (p. 22), *present* (p. 8), *doingOp* (p. 16), *waitingFinishAdminOp* (p. 24)

FS.Enclave.StartUpdateConfigWaitingFloppy

We wait indefinitely for a floppy to be present.

StartUpdateConfigWaitingFloppy

$$
\begin{array}{l}
AdminOpStartedContext \\
\\
\Xi Config \\
\Xi Admin \\
\Xi Floppy \\
\Xi DoorLatchAlarm \\
\hline
the\ currentAdminOp = updateConfigData \\
floppyPresence = absent \\
\\
currentScreen'.screenMsg = insertConfigData \\
currentDisplay' = currentDisplay \\
\\
enclaveStatus' = enclaveStatus
\end{array}
$$

▷ See: *AdminOpStartedContext* (p. 70), *Config* (p. 19), *Admin* (p. 22), *Floppy* (p. 23), *DoorLatchAlarm* (p. 22), *the* (p. 8), *updateConfigData* (p. 22), *absent* (p. 8), *insertConfigData* (p. 16)

$$
\begin{array}{l}
StartUpdateConfigData \mathrel{\widehat=} StartUpdateConfigOK \lor StartUpdateConfigWaitingFloppy \\
\qquad \lor [\, BadAdminLogout \mid enclaveStatus = waitingStartAdminOp \\
\qquad\qquad \land\ the\ currentAdminOp = updateConfigData \,]
\end{array}
$$

▷ See: *StartUpdateConfigOK* (p. 76), *StartUpdateConfigWaitingFloppy* (p. 76), *BadAdminLogout* (p. 63), *waitingStartAdminOp* (p. 24), *the* (p. 8), *updateConfigData* (p. 22)

7.8.2 Storing Configuration Data

FS.Enclave.FinishUpdateConfigDataOK

ScConfig.Suc.Config *ScConfig.Suc.Audit*

The supplied data will be used to replace the current configuration data if it is valid configuration data.

FinishUpdateConfigDataOK

$$
\begin{array}{l}
AdminOpFinishContext \\
\\
\Xi Floppy \\
\Xi DoorLatchAlarm \\
\\
AddElementsToLog \\
\hline
the\ currentAdminOp = updateConfigData \\
\\
currentFloppy \in \mathrm{ran}\ configFile \\
\\
\theta Config' = configFile^{\sim} currentFloppy \\
\\
currentScreen'.screenMsg = requestAdminOp
\end{array}
$$

▷ See: *AdminOpFinishContext* (p. 70), *Floppy* (p. 23), *DoorLatchAlarm* (p. 22), *AddElementsToLog* (p. 33), *the* (p. 8), *updateConfigData* (p. 22), *configFile* (p. 16), *Config* (p. 19)

FS.Enclave.FinishUpdateConfigDataFail

ScConfig.Fail.Read

If the supplied data is not valid configuration data the operation terminates without changing the TIS configuration data.

FinishUpdateConfigDataFail

$$
\begin{array}{l}
\textit{AdminOpFinishContext} \\
\\
\Xi \textit{Config} \\
\Xi \textit{Floppy} \\
\Xi \textit{DoorLatchAlarm} \\
\\
\textit{AddElementsToLog} \\
\hline
\textit{the}\ \textit{currentAdminOp} = \textit{updateConfigData} \\
\\
\textit{currentFloppy} \notin \operatorname{ran} \textit{configFile} \\
\\
\textit{currentScreen}'.\textit{screenMsg} = \textit{invalidData}
\end{array}
$$

▷ See: *AdminOpFinishContext* (p. 70), *Config* (p. 19), *Floppy* (p. 23), *DoorLatchAlarm* (p. 22), *AddElementsToLog* (p. 33), *the* (p. 8), *updateConfigData* (p. 22), *configFile* (p. 16), *invalidData* (p. 16)

$$
\begin{array}{l}
\textit{FinishUpdateConfigData} \mathrel{\widehat{=}} \textit{FinishUpdateConfigDataOK} \lor \textit{FinishUpdateConfigDataFail} \\
\qquad \lor [\, \textit{BadAdminLogout} \mid \textit{enclaveStatus} = \textit{waitingFinishAdminOp} \\
\qquad\qquad \land\ \textit{the}\ \textit{currentAdminOp} = \textit{updateConfigData}\,]
\end{array}
$$

▷ See: *FinishUpdateConfigDataOK* (p. 77), *FinishUpdateConfigDataFail* (p. 78), *BadAdminLogout* (p. 63), *waitingFinishAdminOp* (p. 24), *the* (p. 8), *updateConfigData* (p. 22)

7.8.3 The complete update configuration data operation

FS.Enclave.TISUpdateConfigDataOp

Combining the start and finish phase of this operation gives the complete operation.

$$\textit{TISUpdateConfigDataOp} \mathrel{\widehat{=}} \textit{StartUpdateConfigData} \lor \textit{FinishUpdateConfigData}$$

▷ See: *StartUpdateConfigData* (p. 77), *FinishUpdateConfigData* (p. 78)

7.9 Shutting Down the ID Station

Shutting down the ID Station is a single phase operation.

When the ID Station is shutdown the door is automatically locked so the system is in a secure state. The ID Station cannot be shutdown if the door is currently open, this prevents the enclave being left in an insecure state once TIS is shutdown.

FS.Enclave.ShutdownOK

ScShutdown.Ass.LoggedOn
ScShutdown.Suc.Shutdown
ScShutdown.Suc.Secure
ScShutdown.Suc.Audit
ScShutdown.Con.NonInterleave

ShutdownOK

$$
\begin{array}{l}
AdminOpContext \\
\\
\Xi Config \\
\Xi Floppy \\
\\
AddElementsToLog \\
LockDoor \\
AdminLogout \\
\hline
enclaveStatus = waitingStartAdminOp \\
the\, currentAdminOp = shutdownOp \\
currentDoor = closed \\
\\
currentScreen'.screenMsg = clear \\
\\
enclaveStatus' = shutdown \\
currentDisplay' = blank
\end{array}
$$

▷ See: *AdminOpContext* (p. 69), *Config* (p. 19), *Floppy* (p. 23), *AddElementsToLog* (p. 33), *LockDoor* (p. 38), *AdminLogout* (p. 41), *waitingStartAdminOp* (p. 24), *the* (p. 8), *shutdownOp* (p. 22), *closed* (p. 15), *clear* (p. 16), *blank* (p. 15)

FS.Enclave.ShutdownWaitingDoor

TIS waits indefinitely for the door to be closed before completing the shutdown.

ShutdownWaitingDoor

$$
\begin{array}{l}
AdminOpContext \\
\\
\Xi Config \\
\Xi Floppy \\
\Xi DoorLatchAlarm \\
\Xi Admin \\
\hline
enclaveStatus = waitingStartAdminOp \\
the\, currentAdminOp = shutdownOp \\
currentDoor = open \\
\\
currentScreen'.screenMsg = closeDoor \\
\\
enclaveStatus' = enclaveStatus \\
currentDisplay' = currentDisplay
\end{array}
$$

▷ See: *AdminOpContext* (p. 69), *Config* (p. 19), *Floppy* (p. 23), *DoorLatchAlarm* (p. 22), *Admin* (p. 22), *waitingStartAdminOp* (p. 24), *the* (p. 8), *shutdownOp* (p. 22), *open* (p. 15), *closeDoor* (p. 16)

FS.Enclave.TISShutdownOp

There is nothing that can go wrong with the shutdown operation. This is the only operation that is not prevented by tearing the admin token, as soon as the door is closed TIS will shut down.

$$TISShutdownOp \mathrel{\widehat=} ShutdownOK \lor ShutdownWaitingDoor$$

▷ See: *ShutdownOK* (p. 78), *ShutdownWaitingDoor* (p. 79)

7.10 Unlocking the Enclave Door

Unlocking the enclave door is a single phase operation.

FS.Enclave.OverrideDoorLockOK

ScUnlock.Ass.LoggedOn
ScUnlock.Ass.Quiescent
ScUnlock.Suc.UserIn
ScUnlock.Suc.Audit
ScUnlock.Con.NoInterleave

A guard may need to open the enclave door to admit someone who cannot be admitted by the system.

OverrideDoorLockOK

$$
\begin{array}{l}
AdminOpStartedContext \\
\\
\Xi Floppy \\
\Xi Config \\
\\
AddElementsToLog \\
AdminFinishOp \\
UnlockDoor \\
\hline
the\ currentAdminOp = overrideLock \\
\\
currentScreen'.screenMsg = requestAdminOp \\
currentDisplay' = doorUnlocked \\
\\
enclaveStatus' = enclaveQuiescent
\end{array}
$$

▷ See: *AdminOpStartedContext* (p. 70), *Floppy* (p. 23), *Config* (p. 19), *AddElementsToLog* (p. 33), *AdminFinishOp* (p. 42), *UnlockDoor* (p. 38), *the* (p. 8), *overrideLock* (p. 22), *doorUnlocked* (p. 15)

FS.Enclave.TISUnlockDoorOp

This operation has no failures other than the administrator tearing their token before the operation completes.

$$
\begin{array}{l}
TISOverrideDoorLockOp \mathrel{\widehat=} OverrideDoorLockOK \\
\qquad \lor [\, BadAdminLogout \mid enclaveStatus = waitingStartAdminOp \\
\qquad\qquad \land\ the\ currentAdminOp = overrideLock \,]
\end{array}
$$

▷ See: *OverrideDoorLockOK* (p. 80), *BadAdminLogout* (p. 63), *waitingStartAdminOp* (p. 24), *the* (p. 8), *overrideLock* (p. 22)

8 THE INITIAL SYSTEM AND STARTUP

8.1 The Initial System

FS.TIS.InitIDStation
FMT_MSA.3.1

After initial installation the system has the following properties

- an empty key store, which means it is unable to authorise entry to anyone;
- default configuration data, which does not permit entry to anyone;
- the door latched;
- an empty audit log;
- the internal times all set to zero (a time before the current time).

The door is assumed closed at initialisation, this ensures that the alarm will not sound before the first time that data is polled.

InitDoorLatchAlarm
DoorLatchAlarm

$$
\begin{aligned}
&currentTime = zeroTime \\
&currentDoor = closed \\
&latchTimeout = zeroTime \\
&alarmTimeout = zeroTime
\end{aligned}
$$

▷ See: *DoorLatchAlarm* (p. 22), *zeroTime* (p. 8), *closed* (p. 15)

There are no keys held by the system and the TIS does not know its name, this is supplied as part of enrolment.

InitKeyStore
KeyStore

$$
\begin{aligned}
&issuerKey = \varnothing \\
&ownName = nil
\end{aligned}
$$

▷ See: *KeyStore* (p. 21), *nil* (p. 8)

This default configuration assumes the lowest classification possible for the enclave. This ensures that it does not give inadvertently high clearance to the authorisation certificate. The *authPeriod* and *entryPeriod* functions are set to enable a *securityOfficer* to enter the enclave and re-configure the TIS. This configuration will allow Auth Certificates to be generated with a validity of 2 hours fron the point of issue (assuming that the unit of time is 1/10 sec).

InitConfig

Config

$$
\begin{aligned}
&\mathit{alarmSilentDuration} = 10 \\
&\mathit{latchUnlockDuration} = 150 \\
&\mathit{tokenRemovalDuration} = 100 \\
&\mathit{enclaveClearance.class} = \mathit{unmarked} \\
&\mathit{authPeriod} = \mathit{PRIVILEGE} \times \{\{t : \mathit{TIME} \bullet t \mapsto t \mathinner{..} t + 72000\}\} \\
&\mathit{entryPeriod} = \mathit{PRIVILEGE} \times \{\mathit{CLASS} \times \{\mathit{TIME}\}\}
\end{aligned}
$$

▷ See: *Config* (p. 19), *unmarked* (p. 8), *PRIVILEGE* (p. 9), *TIME* (p. 8), *CLASS* (p. 8)

Initially no administrator is logged on and no administator operations are taking place.

InitAdmin

Admin

$$
\begin{aligned}
&\mathit{rolePresent} = \mathit{nil} \\
&\mathit{currentAdminOp} = \mathit{nil}
\end{aligned}
$$

▷ See: *Admin* (p. 22), *nil* (p. 8)

Initially the statistics are set to zero, indicating no use of the system to date.

InitStats

Stats

$$
\begin{aligned}
&\mathit{successEntry} = 0 \\
&\mathit{failEntry} = 0 \\
&\mathit{successBio} = 0 \\
&\mathit{failBio} = 0
\end{aligned}
$$

▷ See: *Stats* (p. 21)

The initial audit Log is empty and there is no audit alarm.

InitAuditLog

AuditLog

$$
\begin{aligned}
&\mathit{auditLog} = \varnothing \\
&\mathit{auditAlarm} = \mathit{silent}
\end{aligned}
$$

▷ See: *AuditLog* (p. 20), *silent* (p. 15)

Entities that model the real world and are polled and have no security implications are not set at initialisation, these will be updated at the first poll of the real world entities.

Initially the screen and the display are clear and the internal state is *notEnrolled*.

InitIDStation

IDStation

InitDoorLatchAlarm
InitConfig
InitKeyStore
InitStats
InitAuditLog
InitAdmin

$currentScreen.screenMsg = clear$

$currentDisplay = blank$
$enclaveStatus = notEnrolled$
$status = quiescent$

▷ See: *IDStation* (p. 25), *InitDoorLatchAlarm* (p. 81), *InitConfig* (p. 81), *InitKeyStore* (p. 81), *InitStats* (p. 82), *InitAuditLog* (p. 82), *InitAdmin* (p. 82), *clear* (p. 16), *blank* (p. 15), *notEnrolled* (p. 24), *quiescent* (p. 24)

8.2 Starting the ID Station

FS.TIS.TISStartup
FPT_FLS.1.1

We assume that some of the state within TIS is persistent through shutdown and some is not. The persistent items are *Config*, *KeyStore* and *AuditLog* all other state components are set at startup. Those values that are polled can take any valid value, we assume for simplicity that they remain unchanged.

StartContext

$\Delta IDStation$
RealWorldChanges

$\Xi Config$
$\Xi KeyStore$

$InitDoorLatchAlarm'$
$InitStats'$
$InitAdmin'$

$\Xi UserToken$
$\Xi AdminToken$
$\Xi Finger$
$\Xi Floppy$
$\Xi Keyboard$

▷ See: *IDStation* (p. 25), *RealWorldChanges* (p. 27), *Config* (p. 19), *KeyStore* (p. 21), *InitDoorLatchAlarm* (p. 81), *InitStats* (p. 82), *InitAdmin* (p. 82), *UserToken* (p. 23), *AdminToken* (p. 23), *Finger* (p. 23), *Floppy* (p. 23), *Keyboard* (p. 24)

In the case that TIS does not have an allocated name the ID station is assumed to require enrolment.

StartNonEnrolledStation

StartContext

$ownName = nil$

$currentScreen'.screenMsg = insertEnrolmentData$

$currentDisplay' = blank$
$enclaveStatus' = notEnrolled$
$status' = quiescent$

$(\exists\, newElements : \mathbb{F}\, Audit;\ startUnenrolledTISElement : Audit \bullet AddElementsToLog$
$\quad \land\ startUnenrolledTISElement \in newElements)$

▷ See: *StartContext* (p. 83), *nil* (p. 8), *blank* (p. 15), *notEnrolled* (p. 24), *quiescent* (p. 24), *AddElementsToLog* (p. 33)

▷ The *startUnenrolledTISElement* is the audit entry recording that the TIS has been started and requires enrolment.

In the case that TIS does have an allocated name the ID station is assumed to have been previously enrolled.

StartEnrolledStation

StartContext

$ownName \neq nil$

$currentScreen'.screenMsg = welcomeAdmin$

$currentDisplay' = welcome$
$enclaveStatus' = enclaveQuiescent$
$status' = quiescent$

$(\exists\, newElements : \mathbb{F}\, Audit;\ startEnrolledTISElement : Audit \bullet AddElementsToLog$
$\quad \land\ startEnrolledTISElement \in newElements)$

▷ See: *StartContext* (p. 83), *nil* (p. 8), *welcomeAdmin* (p. 16), *welcome* (p. 15), *quiescent* (p. 24), *AddElementsToLog* (p. 33)

▷ The *startEnrolledTISElement* is the audit entry recording that an enrolled TIS has been started.

The complete startup operation is given by:

$TISStartup \mathrel{\widehat{=}} StartEnrolledStation \lor StartNonEnrolledStation$

▷ See: *StartEnrolledStation* (p. 84), *StartNonEnrolledStation* (p. 83)

9 THE WHOLE ID STATION

9.1 Startup

When the TIS is powered up it needs to establish whether it is enrolled or not. This is formally described by

TISStartUp

9.2 The main loop

FS.TIS.TISMainLoop

The TIS achieves its function by repeatedly performing a number of activities within a main loop.

The main loop is broken down into several phases:

- *Poll* - Polling reads the simple real world entities (door, time) and the reads the complex entities (user token reader, admin token reader, fingerprint reader, floppy).
- *Early Updates* - Critical updates of the door latch and alarm are performed as soon as new polled data is available.
- *TIS processing* - TIS processing is the activity performed by TIS, this is influenced by the current *status* of TIS and the recently read inputs.
- *Updates* - Critical updates of the door latch and alarm are repeated once the processing is complete to ensure any internal state changes result in the latch and alarm being set correctly. Less critical updates of the screen and display are also performed once the processing is complete.

9.2.1 Polling

The polling activity is captured by the schema:

TISPoll

9.2.2 Early Updates

The early updates, which only update security critical outputs, are described by:

TISEarlyUpdate

9.2.3 Processing

The the TIS processing depends on the current internal *status* and *enclaveStatus*.

Initially the only activity that can be performed is enrolment, formally captured as *TISEnrol*.

When it is in a *quiescent* state it can start a number of activities. These are started by either reading a user token, an adminstrator token or keyboard data. In addition an administrator may logoff.

Formally the quiecent activities are:

$$TISReadUserToken \lor TISReadAdminToken \lor TISStartAdminOp \lor TISAdminLogout$$

▷ *TISReadUserToken* and *TISReadAdminToken* are the first stages of *TISUserEntry* and *TISAdminLogon*.

Alternatively there is no token present, and no-one logged on, in this case TIS is idle.

TISIdle

$$\Xi IDStation$$
$$\Xi TISControlledRealWorld$$

$$status = quiescent$$
$$enclaveStatus = enclaveQuiescent$$
$$userTokenPresence = absent$$
$$adminTokenPresence = absent$$
$$rolePresent = nil$$

▷ See: *IDStation* (p. 25), *TISControlledRealWorld* (p. 17), *quiescent* (p. 24), *absent* (p. 8), *nil* (p. 8)

Once a user token has been presented to TIS the only activities that can be performed are stages in the multi-phase user entry authentication operation, formally captured as *TISUserEntry*. Since the user entry process is long lived it is necessary to check whether the admin token has been removed during each stage of this operation and act accordingly.

Once an administrator token has been presented to TIS the administrator is logged onto the ID Station, formally captured as *TISAdminLogon*. Having logged the administrator on TIS returns to a *quiescent* state waiting for the administrator to perform an operation, without preventing user entry.

Once an operation request has been made by a logged on administrator TIS performs the, potentially multi-phase, administrator operation, formally captured as *TISAdminOp* captured below:

$$TISAdminOp \mathrel{\widehat=} TISOverrideDoorLockOp \lor TISShutdownOp \lor TISUpdateConfigDataOp \lor TISArchiveLogOp$$

▷ See: *TISOverrideDoorLockOp* (p. 80), *TISShutdownOp* (p. 79), *TISUpdateConfigDataOp* (p. 78), *TISArchiveLogOp* (p. 76)

The overall processing activity is described by:

$$TISProcessing \mathrel{\widehat=} (TISEnrolOp \lor TISUserEntryOp \lor TISAdminLogon \lor TISStartAdminOp \lor TISAdminOp \lor TISAdminLogout \lor TISIdle) \land LogChange$$

▷ See: *TISEnrolOp* (p. 62), *TISUserEntryOp* (p. 57), *TISAdminLogon* (p. 68), *TISStartAdminOp* (p. 73), *TISAdminOp* (p. 86), *TISAdminLogout* (p. 69), *TISIdle* (p. 86), *LogChange* (p. 37)

9.2.4 Final Updates

The updates performed following processing are described by:

TISUpdate

A APPENDIX: READING Z, A SMALL INTRODUCTION

In this section we explain the basics of how to read Z.

The main building block in Z is a *schema*. A Z schema takes the form of a number of state components and, optionally, constraints on the state.

SchemaName

$$\begin{array}{|l} \textit{declarations} \\ \hline \textit{constraints} \\ \hline \end{array}$$

For example we might declare a counter with an upper bound. The counter variable x is declared and it is constrained to be less than 100.

Counter

$$\begin{array}{|l} x : \mathbb{N} \\ \hline x < 100 \\ \hline \end{array}$$

Within the declarative part of a schema we can include schema names. The effect of this inclusion is to bring into scope all the variables and constraints of the schemas that have been included. So in the following *NewCounter* is a counter with a lower bound as well as the upper bound inherited from *Counter*.

NewCounter

$$\begin{array}{|l} \textit{Counter} \\ \hline 40 < x \\ \hline \end{array}$$

▷ See: *Counter* (p. 87)

Schemas are used to describe behaviour under change. Using a convention of decorating state with a $'$ after change we can describe an effect of an operation by describing the new values of the state in terms of its relationship to the old value of the state.

We can describe a simple increment of our counter by the following schema in which the new value of x is the old value of x incremented by 1. Note that the underlying constraints on the variables from the *Counter* schema still apply, so $x' < 100$ is still true.

IncrementCounter

$$\begin{array}{|l} \textit{Counter} \\ \textit{Counter}' \\ \hline x' = x + 1 \\ \hline \end{array}$$

▷ See: *Counter* (p. 87)

In general, instead of writing $\textit{Counter}, \textit{Counter}'$ in our schema declaration we make use of the definition

$\Delta Counter$

$Counter$
$Counter'$

▷ See: *Counter* (p. 87)

the Δ indicating a change to the variables in the schema *Counter*.

Another useful definition is $\Xi Counter$ which is describes the case where the state of the schema is unchanged

$\Xi Counter$

$\Delta Counter$

$\theta Counter = \theta Counter'$

▷ See: *Counter* (p. 87)

▷ The θ here is read as "the state of ".

In addition to schemas, Z allows us to define basic types which give the types of the basic components of our schemas. We might want to introduce the concept of a "string" as a basic type in our system, this will appear as:

$[STRING]$

We can define constants and functions. Here we define a constant *clearString* and print function that turns natural numbers into strings.

$clearString : STRING$
$printNat : \mathbb{N} \rightarrowtail STRING$

Where we know all the possible entities of a basic type we can declare it as a free type. *NEWSTRING* is a free type with a named value element *clearNewString* and a function, *newPrintNat* returning elements of type *NEWSTRING*.

$NEWSTRING ::= clearNewString \mid newPrintNat\langle\langle \mathbb{N} \rangle\rangle$

A key property of the free type is that all elements of the free type are distinct.

These ideas, along with a basic appreciation of predicate logic, should be sufficient to aid reading this specification. For a more detailed description of the Z notation refer to [1].

B APPENDIX: COMMENTARY ON THIS SPECIFICATION

This specification is intended to give a representative formal specification of a realistic system. Budgetary restricitions have meant that the number of administrative operations have been kept to a minimum, although we intend that sufficient have been provided to make the specification representative.

B.1 The structure of the Z

Throughout the specification care has been taken to ensure that each schema is relatively simple. This is an important characteristic in ensuring that a reader can understand the purpose of each schema. Excessive complexity risks making the specification obscure. Schema composition has been used to build up complex operations from simple schemas.

This Z specification is larger than was originally anticipated. We have considered the reason for this and conclude that it is because

- the functionality is larger than originally expected (especially at the administrative interface).
- The core TIS has a fairly large number of interfaces to its environment, two card readers, biometric verifier, door, latch, alarm, internal and external display, floppy disk and keyboard. Each of these has been modelled and the formal notation requires us to explicitly describe what happens to each of these during every operation.
- The entry process contains more steps than originally expected - each interaction with an interface requires a new step.

The relatively large number of interfaces and the desire to compose the system from a number of relatively simple schemas has resulted in the size of specification presented here.

B.2 Issues

A few issues arose while writing this specification; this is expected when formalizing requirements that are stated in any natural language.

We present the more interesting observations here:

B.2.1 The choice of Real World Model

The way in which the real world was modelled is interesting. We would conventionally use Z *inputs*? and *outputs*!, however, these are global through schema composition, which means that one cannot compose two schemas with common inputs or outputs without constraining these entities to be the same.

Within our main loop we want to be able to update the alarm and latch twice; once directly after polling and once after the main processing. Avoiding *inputs*? and *outputs*! allows us to reason about the main loop as a composition of polling, calculations and updates.

This has resulted in us defining the real world using a state schema *RealWorld* and modelling the possible changes to this state. This gives a model that is easier to reason about formally. In particular we can sensibly consider the effect of composing several iterations around the main loop.

We are also able to state and reason about security properties involving real world entities.

B.2.2 Denial of service

When this specification was written the assumption was taken that only one operation could be performed at a time, so once a user had started to attempt authentication and entry in to the enclave, no administrive functions could be supported within the enclave. This assumption was first introduced in the SRS [2]. With this assumption it seemed natural that there only need be one internal state component tracking what the system was doing.

The model does not cover the details of how a token interacts with the card reader, in particular there is no modelling of the Answer-to-Reset (ATR), which is transmitted when a token is first presented to a reader. This was a deliberate abstraction. Within the model we capture the presence of a token and the values held on the token. For operations that are triggered by the presence of a token we allow the operation to start if the system is quiescent and the token is present. One such operation is the user authentication and entry operation. We also require that the token is removed in order for the user authentication and entry process to be considered complete. Otherwise, the system would countinually reprocess a token with bad data until the point that it is removed. The result of this modelling assumption was that we needed an internal state where the system was waiting for a token to be removed following a failure.

The result of having only one internal state component was that placing an invalid token in the reader outside the enclave and leaving it there would block any adminstrator use of the system. Similarly leaving an invalid token in the reader inside the enclave and leaving it there would block any attempt by a user to enter the enclave. This was an unacceptable denial of service. A malicious user could lockup the system. To overcome this problem the internal state was divided in two. One part, *status*, manages the multi-phase user authentication and entry process, the other part, *enclaveStatus*, manages all the activities which involve interaction with TIS from within the enclave. The result of this change was that we were able to eliminate the denial of service attack resulting from a token being left in a reader. A token left in a reader now only blocks other activities that would make use of that reader.

There are other points in the model where the system will wait indefinitely, these have been left as they all arise during operations that can only be performed by an authenticated administrator. We make the assumption that an administrator with privileges to perform these operations will not maliciously leave the system waiting for a floppy disk in order to deny user entry.

B.2.3 The Audit Log

The audit log was the most complex component of the system to model. We wanted the model of the log to be abstract within this specification and we wanted to postpone details of the exact elements that would be placed in the log until the formal design. However we did want to capture some of the key points such as the effect of log overflow and the fact that entries were made to the audit log when key events occurred.

We had to take care that the specification was not too tight, for example we need to allow both the specification and the design to add several entries to the log during the course of an operation and the design may introduce more log entries than are captured in the specification. If the model had been too prescriptive in this area then there would be no viable refinement relation allowing additional log entries in the design over those introduced in the specificaiton. To achieve this within this model we allow a number of entries to be added to the log at a time.

This specification only details when the audit log must be updated, it places no restriction on further entries being added than are detailed in this specification. Once all audit entries have been defined in the design we can place further restrictions on when values may or may not be added to the log.

B.2.4 Malicious attack on the audit log

The Protection Profile [3] requires that old entries in the audit log should be overwritten in the event of the audit log becoming full. This has been modelled in this specification. However this functionality raises the possibility of data being erased from the audit log by performing events that are audited. For example a user could replace the audit log with events corresponding to repeated attempts to log on as an administrator. We did consider preventing all operations, other than archiving the log once the *auditAlarm* has been raised, but since this requires the *AuditManager* to be logged on to the system we cannot exclude the administrator logon activity.

The only other obvious solution would be to shutdown TIS once there was a risk of the audit log becoming full. This assumes that there is a mechanism outside of the TIS function for archiving the log. We have therefore left the functionality allowing unlimited overwriting on the grounds that the alarm is sufficient protection.

B.2.5 Relating enclave entry and Auth Cert generation

The final specification presented here uses indepentent configuration information to determine the authorisation period applied to authorisation certificates and the times at which entry to the enclave should be allowed.

Originally we only allowed entry if the current time was within the authorisation period on the certificate. This seemed to confuse the distinct activities of issuing an Authorisation Certificate and allowing user entry. The original restriction can still be achieved by constraints on the configuration data.

We have also decided not to write an authorisation certificate if its authorisation period is empty. This is a requirements issue that was raised during the production of the specification.

B.2.6 Detail postponed until the design

There are a number of points of detail which are not required to express the system functionality at the abstract level presented within this specification.

One example of this is the deliberate omission of the serial number from the formal model of the certificate Id. It was found that for the purpose of describing the functionality of the TIS the serial number of a certificate was irrelevant. This is because there is no need to demonstrate uniqueness of certificate ids. All that is important within this model is that we can deduce who issued a certificate; this enables us to validate the certificate. The serial number will be introduced in the design where it appears in the detailed information that is audited.

B.2.7 Assumptions on Real World behaviour

In order for the specified TIS to function correctly we need to make a number of assumptions on the behaviour of the modelled real world and its interactions with TIS.

The first assumption is made explicitly in Section 4.1 and is a requirement that the time source can be trusted to provide us with time that increases.

The second assumption is more subtle and was uncovered while performing the precondition proof for *TISPoll* (see page 93). The second assumption is that TIS polls the real world sufficiently frequently that it will always observe the absence of a token before it observes the presence of the next token. This assumption ensures that the Tokens in the reader cannot be swapped without TIS noticing. TIS is capable of noticing token tears, the case where a token is removed mid-processing. However the system as specified will not notice a change in the token contents if it can be swapped without TIS detecting the absence of the first token. This assumption ensures that TIS only makes use of a token that it has previously validated and guarantees that errors such as writing an Authorisation certificate to the wrong token do not occur.

It is necessary to validate both these assumptions to ensure that the system implemented from this specification is indeed secure. If, for example, analysis of the second assumption indicates that it is unreasonable then it would be necessary to reconsider the mechanism by which we monitor the token readers. For instance, it might be necessary to introduce an interrupt triggered by the removal of a token.

C APPENDIX: JUSTIFICATION OF PRECONDITIONS

C.1 Properties

We claim the following important properties of the whole system:

There is an initial state:

FS.TIS.State.InitPOB

$\vdash \exists\, InitIDStation \bullet true$

If there is no state that satisfies the system state invariants then other proof obligations become vacously trivial. It is therefore important to demonstrate that an initial state exists.

The start-up operation is total.

FS.TIS.StartUp.PreTotal

$IDStation \vdash \mathrm{pre}\ TISStartUp$

The processing operation is available whenever the system is not in a *shutdown* state.

FS.TIS.Processing.PreExp
The processing is not total, however we can show that it's precondition is no weaker than $enclaveStatus \neq shutdown$. The internal state *shutdown* represents the system once it is not running so we would expect no processing to occur under this circumstance.

$IDStation; RealWorld \mid \neg\,(enclaveStatus = shutdown \land status = quiescent) \vdash \mathrm{pre}\ TISProcessing$

The polling operation is available on the assumption that the tokens are not changed in the reader so fast that TIS misses observing the absence of the first token before detecting the presence of the second token.

We need to know that while TIS is making use of a validated token the token does not change without TIS noticing it has been removed. This can be expressed formally as follows:

WorldChangesSlowly

$RealWorld$
$IDStation$

$status \in \{gotFinger, waitingFinger, waitingUpdateToken, waitingEntry\} \Rightarrow$
$(userToken = currentUserToken \lor userToken = noT)$

$rolePresent \neq nil \Rightarrow (adminToken = currentAdminToken \lor adminToken = noT)$

▷ See: *RealWorld* (p. 18), *IDStation* (p. 25), *gotFinger* (p. 24), *waitingFinger* (p. 24), *waitingUpdateToken* (p. 24), *waitingEntry* (p. 24), *noT* (p. 15), *nil* (p. 8)

FS.TIS.Poll.PreExp
Polling is not total, it relies on changes to the tokens in the token reader occurring sufficiently slowly that the absence of a token is observered before the presence of a second token is observed.

$$\mathit{WorldChangesSlowly} \vdash \mathrm{pre}\ \mathit{TISPoll}$$

The update and early update operations are total.

FS.TIS.EarlyUpdate.PreTotal

$$\mathit{IDStation}; \mathit{RealWorld} \vdash \mathrm{pre}\ \mathit{TISEarlyUpdate}$$

FS.TIS.Update.PreTotal

$$\mathit{IDStation}; \mathit{RealWorld} \vdash \mathrm{pre}\ \mathit{TISUpdate}$$

C.2 Justifications

C.2.1 Justification of FS.TIS.State.InitPOB

FS.TIS.State.InitPOB

$$\vdash \exists\, \mathit{InitIDStation} \bullet \mathit{true}$$

To demonstrate this it is sufficient to show that the state invariants hold with the constraints on the initial values. We also need to ensure that each value is in type, however by simple inspection we can deduce this to be the case.

For all state components that are left free by the initial state schema it is sufficient to ensure that the types are non-empty. This is the case in all places.

We need to consider the invariants on each of the schemas that are used to build *IDStation* as well as the invariants on the overall *IDStation*.

The invariants on *DoorLatchAlarm* completely define *currentLatch* and *currentAlarm*, the values given in *InitDoorLatchAlarm* result in $\mathit{currentLatch} = \mathit{locked}$ and $\mathit{currentAlarm} = \mathit{silenced}$.

The invariants on *Admin* completely define *availableOps*, the values given in *InitAdmin* result in $\mathit{availableOps} = \varnothing$. The remaining constraint is only applicable when $\mathit{currentAdminOp} \neq \mathit{nil}$ so is true by false implication.

The invariants on *Config* can be satisfied by the following arbitrary choice of variable bindings.

$$\langle \mathit{minPreservedLogSize} \Rrightarrow 10, \mathit{alarmThresholdSize} \Rrightarrow 5 \rangle$$

Considering the constraints on the *IDStation* in turn we note that:

- As *status* = *quiescent* first constraint is true by false implication.
- As *rolePresent* = *nil* the second constraint is true by false implication.
- As *enclaveStatus* = *notEnrolled* the third and sixth constraints are true by false implication.
- As *enclaveStatus* = *notEnrolled* and *currentAdminOp* = *nil* the forth constraint reduces to *false* $\Leftrightarrow$ *false*, which is true.
- As $currentAdminOp' = nil$ the fifth constraint is true by false implication.
- the final constraints define the screen elements *screenStats* and *screenConfig*. As *displayStats* and *displayConfigData* are total functions, we can deduce that these constraints hold.

We therefore deduce that an initial state exists.

C.2.2 Justification of FS.TIS.StartUp.PreTotal

FS.TIS.StartUp.PreTotal

$IDStation \vdash \mathrm{pre}\ TISStartUp$

To demonstrate this we need to show that for any initial values held by *IDStation* there is a binding to the variables in $IDStation'$ that preserves the state invariant.

We first note that, from properties of pre and disjunctions and the definition of *TISStartUp*

$$\mathrm{pre}\ TISStartUp \equiv \mathrm{pre}\ StartEnrolledStation \lor \mathrm{pre}\ StartNonEnrolledStation$$

so we can reduce the problem to considering the preconditions of each of these schemas and ensuring that their disjunction is total.

In a formal proof we would need to ensure that each schema preserves all system and subsystem state invariants. Care has been taken in writing this specification to ensure that operations are sufficiently simple that the preservation of state invariants is easy to check.

We claim that

$$\mathrm{pre}\ StartEnrolledStation \equiv [\, IDStation \mid ownName \neq nil \,]$$

$$\mathrm{pre}\ StartNonEnrolledStation \equiv [\, IDStation \mid ownName = nil \,]$$

The required result follows.

We consider pre *StartEnrolledStation* here by way of an example of the type of arguments that are required to deduce the preconditions of an operation schema.

We need to determine the conditions on the initial state that guarantee that a final state exists and this final state satisfies all the state invariants of $IDStation'$

We note that the state components *UserToken*, *AdminToken*, *Finger*, *Floppy*, *Keyboard*, *Config* and *KeyStore* are defined as not changing. We therefore need not consider invariants that only refer to these state components.

The invariants on *DoorLatchAlarm* completely define *currentLatch* and *currentAlarm*, the values given in $InitDoorLatchAlarm'$ result in $currentLatch = locked$ and $currentAlarm = silenced$.

We note that the precondition on *AddElementsToLog* is a requirement that the *newElements* are no older than the elements already in the log. Without loss of generality we can assume the element *startUnenrolledTISElement* satisifies this property so $newElements = \{startUnenrolledTISElements\}$ is a possible solution. Hence the new $auditLog'$ is defined by *AddElementsToLog*.

Considering the constraints on the *IDStation* in turn we note that:

- As $status' = quiescent$ first constraint is true by false implication.
- As $rolePresent' = nil$ the second constraint is true by false implication.
- As $enclaveStatus' = enclaveQuiescent$ we need to note that $ownName \neq nil$ and *KeyStore* is unchanged to deduce the the third constraint holds.
- As $enclaveStatus' = enclaveQuiescent$ and $currentAdminOp' = nil$ the forth constraint reduces to $false \Leftrightarrow false$.
- As $currentAdminOp' = nil$ the fifth constraint is true by false implication.
- As $enclaveStatus' = enclaveQuiescent$ the sixth constraint is true by false implication.
- the final constraints define the screen elements $screenStats'$ and $screenConfig'$. As *displayStats* and *displayConfigData* are total functions so we can deduce that these constraints hold.

So the only constraint is the explicit contraint on the before state of the *StartEnrolledStation*, namely $ownName \neq nil$. Giving the result.

$$\text{pre } StartEnrolledStation \equiv [\, IDStation \mid ownName \neq nil \,]$$

C.2.3 Justification of FS.TIS.Processing.PreExp

FS.TIS.Processing.PreExp
The processing operation is not total, however we can show that it's precondition is no weaker than $enclaveStatus \neq shutdown$. The internal state *shutdown* represents the system once it is not running so we would expect processing not to occur under this circumstance.

$$IDStation; RealWorld \mid \neg\, (enclaveStatus = shutdown \land status = quiescent) \vdash \text{pre } TISProcessing$$

To prove this we rely heavily on the following property:

For any schemas S and T, pre distributes through disjunction

$$\text{pre } (S \lor T) \equiv (\text{pre } S) \lor (\text{pre } T)$$

Expanding the definition of *TISProcessing*.

$$\begin{aligned}&\text{pre } TISProcessing\\ &\quad \equiv \text{pre } ((TISEnrolOp \lor TISUserEntryOp \lor TISAdminLogon\\ &\qquad\qquad \lor TISStartAdminOp \lor TISAdminOp \lor TISAdminLogout \lor TISIdle) \land LogChange)\end{aligned}$$

We should at this point distribute the *LogChange* through the disjunction and consider the precondition of *TISEnrolOp* $\wedge$ *LogChange* etc. However it transpires that *LogChange* does not add any constraints to the preconditions of each of these operations, it only modifies the AuditLog and all operations on the audit log are sufficiently free to allow these modifications in all circumstances.

So we consider the preconditions of each of the components of the operations. We decompose *TISUserEntryOp* (which is constructed as a disjunction) and consider the preconditions of each of the components:

$$\text{pre } \mathit{TISReadUserToken} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{status} = \mathit{quiescent} \wedge \mathit{userTokenPresence} = \mathit{present} \wedge \mathit{enclaveStatus} \in \{\mathit{enclaveQuiescent}, \mathit{waitingRemoveAdminTokenFail}\}]$$

$$\text{pre } \mathit{TISValidateUserToken} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{status} = \mathit{gotUserToken}]$$

$$\text{pre } \mathit{TISReadFinger} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{status} = \mathit{waitingFinger}]$$

$$\text{pre } \mathit{TISValidateFinger} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{status} = \mathit{gotFinger}]$$

$$\text{pre } \mathit{TISWriteUserToken} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{status} = \mathit{waitingUpdateToken}]$$

$$\text{pre } \mathit{TISValidateEntry} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{status} = \mathit{waitingEntry}]$$

$$\text{pre } \mathit{TISUnlockDoor} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{status} = \mathit{waitingRemoveTokenSuccess}]$$

$$\text{pre } \mathit{TISCompleteFailedAccess} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{status} = \mathit{waitingRemoveTokenFail}]$$

We decompose *TISAdminOp* and consider the preconditions of each of the components.

$$\text{pre } \mathit{TISStartArchiveLog} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{enclaveStatus} = \mathit{waitingStartAdminOp} \wedge \mathit{currentAdminOp} \neq \mathit{nil} \wedge \mathit{currentAdminOp} = \mathit{archiveLog}]$$

$$\text{pre } \mathit{TISFinishArchiveLog} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{enclaveStatus} = \mathit{waitingFinishAdminOp} \wedge \mathit{currentAdminOp} \neq \mathit{nil} \wedge \mathit{currentAdminOp} = \mathit{archiveLog}]$$

$$\text{pre } \mathit{TISStartUpdateConfigData} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{enclaveStatus} = \mathit{waitingStartAdminOp} \wedge \mathit{currentAdminOp} \neq \mathit{nil} \wedge \mathit{currentAdminOp} = \mathit{updateConfigData}]$$

$$\text{pre } \mathit{TISFinishUpdateConfigData} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{enclaveStatus} = \mathit{waitingFinishAdminOp} \wedge \mathit{currentAdminOp} \neq \mathit{nil} \wedge \mathit{currentAdminOp} = \mathit{updateConfigData}]$$

$$\text{pre } \mathit{TISShutdownOp} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{enclaveStatus} = \mathit{waitingStartAdminOp} \wedge \mathit{currentAdminOp} \neq \mathit{nil} \wedge \mathit{currentAdminOp} = \mathit{shutdownOp}]$$

$$\text{pre } \mathit{TISOverrideDoorLockOp} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{enclaveStatus} = \mathit{waitingStartAdminOp} \wedge \mathit{currentAdminOp} \neq \mathit{nil} \wedge \mathit{currentAdminOp} = \mathit{overrideLock}]$$

From these and the *IDStation* invariants:

$$\mathit{enclaveStatus} \in \{\mathit{waitingStartAdminOp}, \mathit{waitingFinishAdminOp}\} \Leftrightarrow \mathit{currentAdminOp} \neq \mathit{nil}$$

$$(\mathit{currentAdminOp} \neq \mathit{nil} \wedge \mathit{the}\, \mathit{currentAdminOp} \in \{\mathit{shutdownOp}, \mathit{overrideLock}\}) \Rightarrow \mathit{enclaveStatus} = \mathit{waitingStartAdminOp}$$

we can deduce

$$\text{pre } \mathit{TISAdminOp} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{enclaveStatus} \in \{\mathit{waitingStartAdminOp}, \mathit{waitingFinishAdminOp}\}]$$

We decompose *TISEnrol* and consider the preconditions of each of the components.

$$\text{pre } \mathit{ReadEnrolmentData} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{enclaveStatus} = \mathit{notEnrolled}]$$

$$\text{pre } \mathit{ValidateEnrolmentData} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{enclaveStatus} = \mathit{waitingEnrol}]$$

$$\text{pre } \mathit{CompleteFailedEnrolment} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{enclaveStatus} = \mathit{waitingEndEnrol}]$$

We decompose *TISAdminLogon* and consider the preconditions of each of the components.

$$\text{pre } \mathit{TISReadAdminToken} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{enclaveStatus} = \mathit{enclaveQuiescent} \land \mathit{adminTokenPresence} = \mathit{present} \land \mathit{status} \in \{\mathit{quiescent}, \mathit{waitingRemoveTokenFail}\}]$$

$$\text{pre } \mathit{TISValidateAdminToken} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{enclaveStatus} = \mathit{gotAdminToken}]$$

$$\text{pre } \mathit{TISCompleteFailedAdminLogon} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{enclaveStatus} = \mathit{waitingRemoveAdminTokenFail}]$$

We decompose *TISAdminLogout* and consider the preconditions of each of the components.

$$\text{pre } \mathit{TokenRemovedAdminLogout} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{rolePresent} \neq \mathit{nil} \land \mathit{enclaveStatus} = \mathit{enclaveQuiescent} \land \mathit{adminTokenPresence} = \mathit{abscent}]$$

$$\text{pre } \mathit{AdminTokenTimeout} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{rolePresent} \neq \mathit{nil} \land \mathit{enclaveStatus} = \mathit{enclaveQuiescent} \land \mathit{adminTokenPresence} = \mathit{present} \land \neg\, \mathit{AdminTokenOK}]$$

$$\text{pre } \mathit{TISCompleteTimeoutAdminLogout} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{enclaveStatus} = \mathit{waitingRemoveAdminTokenFail}]$$

The remaining operations are *TISStartAdminOp, TISAdminLogout* and *TISIdle*:

$$\text{pre } \mathit{TISStartAdminOp} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{rolePresent} \neq \mathit{nil} \land \mathit{enclaveStatus} = \mathit{enclaveQuiescent} \land \mathit{adminTokenPresence} = \mathit{present} \land \mathit{status} \in \{\mathit{quiescent}, \mathit{waitingRemoveTokenFail}\}]$$

$$\text{pre } \mathit{TISIdle} \equiv [\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{rolePresent} = \mathit{nil} \land \mathit{enclaveStatus} = \mathit{enclaveQuiescent} \land \mathit{adminTokenPresence} = \mathit{absent} \land \mathit{status} = \mathit{quiescent}, \land \mathit{userTokenPresence} = \mathit{absent}]$$

By considering properties of conjunction and disjunction we can deduce

$$\text{pre } (\mathit{TISStartAdminOp} \lor \mathit{TISAdminLogout} \lor \mathit{TISIdle}$$

$$\begin{aligned}
&\vee\ \mathit{TISReadUserToken} \vee \mathit{TISReadAdminToken})\\
\equiv\ &[\,\mathit{IDStation}; \mathit{RealWorld} \mid\\
&\quad (\mathit{status} = \mathit{quiescent} \wedge \mathit{enclaveStatus} = \mathit{enclaveQuiescent})\\
&\quad \vee\ (\mathit{status} = \mathit{waitingRemoveTokenFail} \wedge \mathit{enclaveStatus} = \mathit{enclaveQuiescent}\\
&\qquad \wedge\ \mathit{adminTokenPresence} = \mathit{present})\\
&\quad \vee\ (\mathit{status} = \mathit{quiescent} \wedge \mathit{enclaveStatus} = \mathit{waitingRemoveAdminTokenFail}\\
&\qquad \wedge\ \mathit{userTokenPresence} = \mathit{present})\\
&\quad \vee\ (\mathit{enclaveStatus} = \mathit{enclaveQuiescent} \wedge \mathit{rolePresent} \neq \mathit{nil}\\
&\qquad \wedge\ \mathit{adminTokenPresence} = \mathit{absent})\\
&\quad \vee\ (\mathit{enclaveStatus} = \mathit{enclaveQueiscent} \wedge \mathit{rolePresent} \neq \mathit{nil}\\
&\qquad \wedge\ \mathit{adminTokenPresence} = \mathit{present} \wedge \neg\, \mathit{AdminTokenOK}\,]
\end{aligned}$$

By considering the coverage of the values of *status* and *enclaveStatus* we can deduce

$$\begin{aligned}
&\mathrm{pre}\ (\mathit{TISEnrolOp} \vee \mathit{TISUserEntryOp} \vee \mathit{TISAdminLogon}\\
&\qquad \vee\ \mathit{TISStartAdminOp} \vee \mathit{TISAdminOp} \vee \mathit{TISAdminLogout} \vee \mathit{TISIdle})\\
\equiv\ &[\,\mathit{IDStation}; \mathit{RealWorld} \mid\\
&\qquad (\mathit{status} = \mathit{quiescent} \wedge \mathit{enclaveStatus} = \mathit{enclaveQuiescent})\\
&\qquad \vee\ (\mathit{status} \neq \mathit{quiescent})\\
&\qquad \vee\ (\mathit{enclaveStatus} \neq \mathit{enclaveQuiescent} \wedge \mathit{enclaveStatus} \neq \mathit{shutdown})\,]
\end{aligned}$$

rearranging this gives:

$$\begin{aligned}
&\mathrm{pre}\ (\mathit{TISEnrolOp} \vee \mathit{TISUserEntryOp} \vee \mathit{TISAdminLogon}\\
&\qquad \vee\ \mathit{TISStartAdminOp} \vee \mathit{TISAdminOp} \vee \mathit{TISAdminLogout} \vee \mathit{TISIdle}\\
\equiv\ &[\,\mathit{IDStation}; \mathit{RealWorld} \mid \mathit{status} \neq \mathit{quiescent} \vee \mathit{enclaveStatus} \neq \mathit{shutdown}\,]
\end{aligned}$$

We have already argued that *LogChange* does not effect the precondition, and hence the required result follows.

C.2.4 Justification of FS.TIS.Poll.PreExp

FS.TIS.Poll.PreExp
Polling is not total, it relies on changes to the tokens in the token reader occurring sufficiently slowly that the absence of a token is observered before the presence of a second token is observed.

$$\mathit{WorldChangesSlowly} \vdash \mathrm{pre}\ \mathit{TISPoll}$$

To demonstrate this we notice that there are no constraints on the initial state in the *TISPoll* schema. It is thus sufficient to check that all system invariants are maintained. The only invariants that need be checked are those that involve entities that change. There are two of these:

Firstly

$$\begin{aligned}
&\mathit{status} \in \{\, \mathit{gotFinger}, \mathit{waitingFinger}, \mathit{waitingUpdateToken}, \mathit{waitingEntry} \,\} \Rightarrow\\
&\quad ((\exists\, \mathit{ValidToken} \bullet \mathit{goodT}(\theta \mathit{ValidToken}) = \mathit{currentUserToken}')\\
&\qquad \vee\ (\exists\, \mathit{TokenWithValidAuth} \bullet \mathit{goodT}(\theta \mathit{TokenWithValidAuth}) = \mathit{currentUserToken}'))
\end{aligned}$$

This holds under the assumptions of *WorldChangesSlowly* since by the definition of *PollUserToken* we can deduce $\mathit{currentUserToken}' = \mathit{currentUserToken}$ under these conditions.

The second invariant

$$rolePresent \neq nil \Rightarrow (\exists\, TokenWithValidAuth \bullet goodT(\theta TokenWithValidAuth) = currentAdminToken')$$

Again, this holds under the assumptions of *WorldChangesSlowly* since by the definition of *PollAdminToken* we can deduce $currentAdminToken' = currentAdminToken$ under these conditions.

C.2.5 Justification of FS.TIS.EarlyUpdate.PreTotal

FS.TIS.EarlyUpdate.PreTotal

$$IDStation; RealWorld \vdash \text{pre } TISEarlyUpdate$$

To demonstrate this we notice that there are no constraints on the initial state in the *TISEarlyUpdate* schema. It is thus sufficient to check that all system invariants are maintained. The only invariants that need be checked are those that involve entities that change, as there are no invariants involving state changed by this operation we can deduce that the operation is total.

C.2.6 Justification of FS.TIS.Update.PreTotal

FS.TIS.Update.PreTotal

$$IDStation; RealWorld \vdash \text{pre } TISUpdate$$

To demonstrate this we notice that there are no constraints on the initial state in the *TISUpdate* schema. It is thus sufficient to check that all system invariants are maintained. The only invariants that need be checked are those that involve entities that change, as there are no invariants involving state changed by this operation we can deduce that the operation is total.

D APPENDIX: TRACING OF SRS REQUIREMENTS

One of the scenarios has been mapped to the Formal Functional Specification to show the type of mapping that would normally be done for high integrity development. The remaining will be completed if time allows, and if errors found through the mapping process and subsequently suggest it will be cost effective.

D.1 Mapping of: User gains allowed initial access to Enclave

Description

A User who should be allowed access to the enclave is given access, making use of biometric authentication.

Stimulus

User inserts a smartcard into the smartcard reader.

Assumptions

ScGainInitial.Ass.ValidStart
The ID Station has valid start-up data.

This cannot be false, as there is no concept in the specification of non-valid start-up (enrolment) data. In practice,

$$\text{pre } \mathit{TISUserEntryOp} \Rightarrow \mathit{enclaveStatus} \notin \{\mathit{notEnrolled}, \mathit{waitingEnrolled}, \mathit{waitingEndEnrol}\}$$

which implies that enrolment has been carried out successfully.

ScGainInitial.Ass.ValidConfig
The ID Station has a valid data configuration.

This cannot be false, as there is no concept in the specification of non-valid start-up (enrolment) data. In practice,

$$\text{pre } \mathit{TISUserEntryOp} \Rightarrow \mathit{enclaveStatus} \notin \{\mathit{notEnrolled}, \mathit{waitingEnrolled}, \mathit{waitingEndEnrol}\}$$

which implies that enrolment has been carried out successfully.

ScGainInitial.Ass.Quiescent
The ID Station is quiescent (no other access attempts, configuration changes or start-up activities are in progress).

$$\text{pre } \mathit{TISUserEntryOp} \Rightarrow \mathit{enclaveStatus} \in \{\mathit{quiescent}, \mathit{waitingRemoveAdminTokenFail}\}$$

which is a state from which no other access attempts can be in progress, or configuration or start-up activities.

ScGainInitial.Ass.Secure
The User is outside the enclave; the door is closed and locked.

The implementation does not need to make this assumption.

ScGainInitial.Ass.ValidUser
The card inserted by the User has a valid ID Certificate, I&A Certificate, and Privilege Certificate, and the card inserted by the User has a valid fingerprint template that matches the fingerprint of the User's finger.

This, together with the next condition, are both met by:

$$status = waitingRemoveTokenSuccess$$
$$\wedge$$
$$\mathrm{pre}\ BioCheckRequired \Rightarrow \mathrm{post}BioCheckRequired \wedge \mathrm{post}ValidateFingerOK$$

ScGainInitial.Ass.PoorAC
The card inserted by the User does not have a valid, current Authorisation Certificate.

See condition above.

Success End-conditions

ScGainInitial.Suc.UserCard
The User has possession of the card he originally inserted.

$$\mathrm{post}\ UnlockDoor \Rightarrow userTokenPresence = absent$$

ScGainInitial.Suc.GoodAC
The card inserted by the User contains a current, valid Authorisation Certificate with

- *validity time: from now until now+(length of timespecified in ID Station configuration data)*
- *security level: equal to the minimum of (the security level defined in the ID Station configuration data) and (the security level in the Permission Certificate on the card inserted by the User)*

$$status = waitingRemoveTokenSuccess \wedge \mathrm{pre}\ BioCheckRequired \Rightarrow \mathrm{post}WriteUserTokenOK$$

ScGainInitial.Suc.PersistCerts
The card inserted by the User contains the same, unchanged ID Certificate, I&A Certificate, and Privilege Certificate it had at the beginning of the scenario.

All possible sequences of operations to achieve this scenario have Xi UserToken at each stage, except in WriteUserToken, where there are explicit predicates to preserve these certificates.

ScGainInitial.Suc.UserIn
The User is in the Enclave.

post *UnlockDoor* leaves a time interval after the door has been opened. This time interval will only be completed (and the door latched again) as part of *Poll*, and the passage of time. This will allow a user that chooses to, to enter the enclave.

ScGainInitial.Suc.Locked
The Enclave door is closed and locked.

Poll occurs frequently and regularly, and given sufficient time, this will force the timeout on the door (*currentTime* to exceed *latchTimeout*), causing the door to lock (invariant in *DoorLatchAlarm*, and updating the real latch in *TISEarlyUpdate* and *TISUpdate*). The value of latchTimeout is only ever set to currentTime + *latchUnlockDuration*, (in *UnlockDoor*) or currentTime (in *LockDoor*), and so the definition of ”sufficient time” is the value of *latchUnlockDuration*.

ScGainInitial.Suc.Audit
The following events have been recorded in the Audit Log (in any order), and the existing audit records are preserved:

AddElementsToLog is the schema that adds audit events, and it preserves the logs (except when there is an overflow, in which case it conforms to the failure condition SCGainInitial.Fail.AuditPreserve). The occurrences of the individual audit events are given below.

- Insertion of card
  ReadUserToken
- Removal of card
  UserTokenTear, *UnlockDoor*, *FailedAccessTokenRemoved*
- Reading data from card (possibly multiple failures, but at least one success)
  ReadUserToken
- Writing data to the card (possibly multiple failures, but at least one success)
  WriteUserTokenOK
- Reading fingerprint image
  ReadFingerOK
- Setting the door to locked.
  AuditLatch
- Setting the door to unlocked.
  Auditlatch
- Door opening
  AuditDoor
- Door closing
  AuditDoor
- Writing data to the display.
  AuditDisplay
- Validation of any certificate (possibly multiple failures, but at least one success)
  This is not visible at this level, although the success or failure of parts of the process are audited, and there is room in the refinement to add explicit auditing of certificate operations.
- Creation or modification of signed Authorisation certificate
  WriteUserTokenOK
- Comparison of fingerprint image and template (possibly multiple failures, but at least one success)
  ValidateFingerOK

Failure Conditions

All error conditions in the formal functional specification explicitly state Xi on the state (except for the audit part of the state)

ScGainInitial.Fail.ReadCard
The card inserted by the User does not allow all its data to be successfully read, possibly due to being incorrectly inserted in the first place; being a faulty card; having the incorrect information on it; or being removed before all the information has been read. The set of data to be read is at least:

- *ID Certificate*
- *I&A Certificate*
- *Privilege Certificate*
- *Authorisation Certificate*
- *Fingerprint Template (contained in the I&A Certificate)*

ScGainInitial.Fail.Fingerprint
A matching fingerprint has not been read, possibly due to no finger being presented to the fingerprint reader within X seconds of the display requesting a fingerprint; or the fingerprint not being successfully read within X seconds of

the display requesting a fingerprint; or the fingerprint that was successfully read not being successfully matched to the template read from the card. The value X shall be taken from configuration data of the ID Station.

ScGainInitial.Fail.WriteCard
The card originally inserted by the User does not allow a new Authorisation Certificate to be successfully written, possibly due to being incorrectly inserted in the first place; being a faulty card; or being removed before all the information has been written.

ScGainInitial.Fail.UserSlow
The User is too slow in opening the door, so the door locks with the user still outside the enclave. Or the user opens the door, but chooses not to pass through, closing the door again.

Not implemented.

ScGainInitial.Fail.DoorPropped
Once the door has been opened, it is not allowed to close (it is propped open).

Not implemented.

ScGainInitial.Fail.Audit
Audit files cannot be successfully written. Result: the Door is locked and the system is shutdown.

Not currently in the formal specification.

ScGainInitial.Fail.AuditPreserve
Space for audit files has been exhausted. Result: the oldest audit records are overwritten with the new audit records, and an alarm is raised to the Guard.

AddElementsToLog

Constraints

ScGainInitial.Con.NoInterleave
No ID Station restart or Configuration data changes will be allowed during this scenario.

$$status \neq quiescent \Rightarrow \neg \, \mathrm{pre}\ ValidateOpRequestOK$$

D.2 Requirements out of scope

This section lists the requirements from the SRS [2] that are not referenced from this document, with a justification for their omission.

D.2.1 Not Implemented

The following requirements have not been implemented within this formal specification.

These requirements all relate to the action following failure to write to the audit log.

FS.NotInScope.NotImplemented	
ScGainInitial.Fail.Audit	*ScConfig.Fail.Audit*
ScProhibitInitial.Fail.Audit	*ScAudit.Fail.Audit*
ScGainRepeat.Fail.Audit	*ScUnlock.Fail.Audit*
ScStart.Fail.Audit	*ScLogOn.Fail.Audit*
ScShutdown.Fail.Audit	*ScLogOff.Fail.Audit*

D.2.2 User behaviour

The following requirements have not been captured within the formal specification of the software software since they are the result of human behaviour alone.

FS.NotInScope.UserBehaviour	
ScGainInitial.Fail.DoorPropped	*ScGainRepeat.Fail.UserSlow*
ScGainInitial.Fail.UserSlow	*ScUnlock.Fail.DoorPropped*
ScGainRepeat.Fail.DoorPropped	*ScUnlock.Fail.UserSlow*

D.2.3 Assumption of Secure Enclave

The following assumptions are not enforced within the Formal Specification. It was felt unnecessarily restrictive to enforce the assumption that the door was closed and locked prior to commencement of an operation. However guarantees of the resulting security of the enclave following an operation can only be made within the context of the state of the environment at the start of the operation. For example, if the door is open there is nothing to stop a user who does not have a valid token from entering the enclave.

FS.NotInScope.SecureAssumption	
ScGainInitial.Ass.Secure	*ScConfig.Ass.Secure*
ScProhibitInitial.Ass.Secure	*ScAudit.Ass.Secure*
ScGainRepeat.Ass.Secure	*ScUnlock.Ass.Secure*
ScStart.Ass.Secure	*ScLogOn.Ass.Secure*
ScShutdown.Ass.Secure	*ScLogOff.Ass.Secure*

D.2.4 Performance Limitations

Due to performance limitations the system does not read back the user token after writing data to it so we cannot be sure whether the write was successful or not.

FS.NotInScope.PerformanceLimitations
ScGainInitial.Fail.WriteCard

D.3 General Requirements

Several of the requirements are of a general nature and demonstration of their satisfaction by this specification requires analysis of the whole specification rather than reference to a single (or small number of) operation schemas. These are detailed in the following sections.

D.3.1 Valid Start

This cannot be false, there is no concept in this specification of non-valid start-up data. In practice all operations other than enrolment have a precondition which implies

$$enclaveStatus \notin \{notEnrolled, waitingEnrolled, waitingEndEnrol\}$$

from this we can deduce that enrolment has been carried out successfully.

FS.General.ValidStart	
ScGainInitial.Ass.ValidStart	*ScGainRepeat.Ass.ValidStart*
ScProhibitInitial.Ass.ValidStart	

D.3.2 Valid Config

This cannot be false, there is no concept in this specification of non-valid start-up data. In practice all operations other than enrolment have a precondition which implies

$$enclaveStatus \notin \{notEnrolled, waitingEnrolled, waitingEndEnrol\}$$

from this we can deduce that enrolment has been carried out successfully.

FS.General.ValidConfig	
ScGainInitial.Ass.ValidConfig	*ScGainRepeat.Ass.ValidConfig*
ScProhibitInitial.Ass.ValidConfig	

D.3.3 Persistent Certificates

The formal specification indicates that it does not change the user token by the presence of the Ξ*UserToken* on the majority of the operations. The only operation that does not have this constraint is *WriteUserToken*, this operation does not form part of any of the scenarios traced below.

FS.Gereral.PersistCertificates	
ScProhibitInitial.Suc.PersistCerts	*ScGainRepeat.Suc.PersistCerts*

D.3.4 Enclave Security

The formal specification indicates that it does not modify the timer that controls the latch unless the conditions for the partial operations *UnlockDoorOK* or *OverrideDoorLockOK* are satisfied. So if the system was secure prior to the start of a senario then it will be secure at the end for all senarios that do not permit these operations. The following scenarios do not satisfy the preconditions of *UnlockDoorOK* or *OverrideDoorLockOK*.

FS.General.DoorRemainsLocked	
ScProhibitInitial.Suc.UserOut	*ScConfig.Suc.Secure*
ScProbibitInitial.Suc.Locked	*ScLogOn.Suc.Secure*
ScStart.Suc.Secure	*ScLogOff.Suc.Secure*

E APPENDIX: TRACING OF ST REQUIREMENTS

E.1 Mapping of Functional Security Requirements

This section justifies the manner in which this specification satisifies the security requirements presented in the Security Target [4].

SFP.DAC (new table):
Admin invariants define available Ops for each role. *ValidateOpRequestOK* checks the requested op against the role. *ValidateAdminTokenOK* logs the user on based on their Authorisation Certificate. Viewing is controlled by *UpdateScreen*, which defines the information displayed on the screen. It displays configuration data only if the security officer is present. System statistics are deamed non-secure, and are displayed when any administrator is logged on.

FAU_ARP.1.1:
Defined class of alarms are: door open, latched, too long; audit files truncating with loss.
DoorLatchAlarm invariant defines the internal decision to alarm the door. This is effected in the real world with *UpdateAlarm*, done twice on each main loop.
AddElementsToLog and *ClearLog* modify the audit alarm internally. This is effected in the real world with *UpdateAlarm*, done twice on each main loop.
Alarming is audited in *AuditAlarm* and *AuditLogAlarm*, part of *LogChange*. invoked once every main loop.

FAU_GEN.1.1:
Every significant schema audits. A check will need to be done at the code level that the actual collection of events is as desired. Confirm necessary list of audit events only once the other exclusions are in place.

FAU_GEN.1.2:
Free type Audit is not elaborated in this specification. Will need to be checked when elaborated in the design.

FAU_GEN.2.1:
See FAU_GEN.1.2

FAU_SAA.1.1:
Alarming is the mechanism for responding immediately to audited events, and so this requirement is met by the FAU_ARP.1.1 above.

FAU_SAA.1.2:
See FAU_SAA.1.1

FAU_STG.2.3:
AddElementsToLog

FAU_STG.4.1:
AddElementsToLog

FCO_NRO.2.1:
ValidToken defines some of the checks on all the certificates, *TokenWithValidAuthCert* including the Authorisation Certificate. This is invoked in *UserTokenOK* and *UserTokenWithOKAuthCert*, which

add the remaining checks.

FCO_NRO.2.2:
See FCO_NRO.2.1

FCO_NRO.2.3:
See FCO_NRO.2.1

FDP_ACC.1.1:
Access to system objects is controlled by giving admin users roles, and controlling which operations they have access to. This is covered in SFP_DAC. Access to user objects (certificates on tokens) is restricted by the basic design of the system: certificates are read and validated during user entry (*TISUserEntryOp*) for a single user at a time, and then discarded. There is no opportunity for any other user or administrator to access these objects.

FDP_ACF.1.1:
see FDP_ACC.1.1

FDP_ACF.1.2:
see FDP_ACC.1.1

FDP_ACF.1.3:
see FDP_ACC.1.1

FDP_ACF.1.4:
see FDP_ACC.1.1

FDP_DAU.2.1:
This refers to ensuring validity of the users' credentials, i.e. the cryptographic processes that ensure the certificates are signed correctly. This is therefore covered in *ValidToken*, and *TokenWithValidAuthCert* which define some of the checks on all the certificates, including the Authorisation Certificate, and the invoking *UserTokenOK* and *UserTokenWithOKAuthCert*, which add the remaining checks. Ensuring the Authorisation Certificate is valid is carried out in *TISWriteUserToken*. But note that as the cryto is simulated, this validity checking and guarantee is not real.

FDP_DAU.2.2:
See FDP_DAU.2.1

FDP_RIP.2.1:
Not really a functional requirement. But the specification overall does not provide any function to access the previous information in a resource. This requirement is deferred.

FDP_UIT.1.1:
Authorisation Certificates are created by *NewAuthCert*, which ensures they can be validated by the key of the ID Station.

FDP_UIT.1.2:
See FDP_UIT.1.1.

FIA_UAU.2.1:
This is the basic operation of the TIS, and is implemented by the whole behaviour - not permitting user entry until authorisation and not letting administrators carry out operations until they have logged on. This cannot be mapped to any specific part of the specification, but the property is

captured in the formal security policy, and the proof follows from there.

FIA_UAU.3.1:
Only detection of forgery of token information is in scope. This maps to *ValidToken* and the cryptographic check implied by this.

FIA_UAU.3.2:
The design choice of using biometric data means that copying the token is not an effective attack, because it is not possible to copy the other person's actual biometric object (e.g. their finger). Copying the authorisation certificate without copying the token is prevented by *ValidToken*, which checks the token ID in the certificate is the same as on the Token, and *NewAuthCert*, which constructs auth certs with the correct token ID in them. This does depend on the token ID being unique.

FIA_UAU.7.1:
The series of updates to the display given in the *TISUserEntryOp* operation presents information to the user only at major milestones in the authentication process, and so leaks no useful information to the user.

FIA_UID.2.1:
The first action a user must carry out is to present their token, from which the certificates are extracted (*ReadUserToken* and *ReadAdminToken*). These certificates identify the user to the system before they can do anything else.

FIA_USB.1.1:
For user entry, no operation is possible other than to unlock the door, which is done only if the user is allowed entry, so this requirement does not really apply. For administrators, the actions allowed to a user are based on their role, as covered by SFP.DAC.

FMT_MOF.1.1:
This table has been modified by the ST, making it clear that only configuration data will be modifiable, and then only by the security officer. The allocation of operations to the security officer is covered in the Admin invariant, which gives him (and only him) only *updateConfigData*. *ValidateOpRequest* will therefore limit him to modification of the configuration data.

FMT_MSA.1.1:
See SFP.DAC.

FMT_MSA.2.1:
Configuration data and enrolment data, the only security information, is entered via a floppy. The definition of the possible content of a floppy includes $enrolmentFile\langle\langle ValidEnrol\rangle\rangle$, $configFile\langle\langle Config\rangle\rangle$ and *badFloppy*. The invariants on *ValidEnrol* and *Config* ensure that if the floppy is regarded as having enrolment or configuration data, it will be correctly constructed (secure). If the data is not correct, the floppy will be regarded as having *badFloppy* data, and *EnrolmentDataOK* and *FinishUpdateConfigDataOK* will not read it. (Invariants will need to be defined for all security values in *Config* and *ValidEnrol* in discussion with the client).

FMT_MSA.3.1:
InitIDStation

FMT_MSA.3.2:
No information is created by the Security Officer, and so this requirement does not need to be implemented.

FMT_MTD.1.1:
See SFP.DAC

FMT_MTD.3.1:
See FMT_MSA.2.1

FMT_SAE.1.1:
See SFP.DAC.

FMT_SAE.1.2:
Not implemented. Behaviourally, whether the authorisation cert is there and out of validity, or not there, is invisible to the user.

FMT_SMR.2.1:
See SFP.DAC

FMT_SMR.2.2:
ValidateAdminTokenOK associates a user with the role given to them in their authorisation certificate.

FMT_SMR.3.1:
The Administrator must insert their admin token and have it read by the system (*ReadAdminTokenOK*) before they are given any admin role.

FPT_FLS.1.1:
Power failure or crash will require the system to be re-started. There is no persistent state, and a re-start will require the system to be re-started in a secure state, as with any start-up.

FPT_RVM.1.1:
The invariant in *DoorLatchAlarm* ensures that the latch never remains unlocked beyond the time set by the TIS. It also ensures that the alarm will sound if the door remains open for too long. The variables *status* and *enclaveStatus* act as state-machine controls, and ensure that each operation can only be performed when the system is in the correct state.

FPT_STM.1.1:
PollTime reads the time from an external source (assumed reliable). This is done once per main loop.

FRU_PRS.1.1:
The variables *status* and *enclaveStatus* act as state-machine controls and ensure that once User Entry or Admin validation have started, they continue to appropriate points, ignoring other accesses. This in effect assures that an action in hand has priority over new actions - new subjects have lower priority to old subjects.

FRU_PRS.1.2:
See FRU_PRS.1.1

E.2 Requirements out of scope

This section lists the requirements that are not referenced from this document, with a justification for their omission.

E.2.1 Deferred

The following requirements have been deferred to the design due to their detailed nature.

FS.NotInScope.Deferred	
FAU_GEN.1.2	*FDP_RIP.2.1*
FAU_GEN.2.1	

E.2.2 NotImplemented

The following requirements have not been implemented, the justification for their non-implementation is covered in section E.1.

FS.NotInScope.STNotImplemented
FMT_SAE.1.2

E.3 General Requirements

Several of the security requirements are of a general nature and demonstration of their satisfaction by this specification requires analysis of the whole specification rather than reference to a single (or small number of) operation schemas. The justification of satisfaction of each of these requirements is presented in section E.1.

FS.General.STRequirements	
FRU_PRS.1.1	*FRU_PRS.1.2*

F APPENDIX: Z INDEX

This section contains an index of Z terms. This contains all the Z schemas, types and functions defined in the specification.

G APPENDIX: TRACEUNIT INDEX

An index of traceunits. This contains all the traceunits placed in the specification to enable the elements of the specification to be traced to the design.

H APPENDIX: REQUIREMENTS INDEX

An index of traceunits. This contains all the traceunits in the requirements documents . All requirements are listed with the pages from which they are referenced.