Tokeneer ID Station
Code Verification Summary

S.P1229.52.1
Issue: 1.2
Status: Definitive
25th August 2008

Originator

Janet Barnes (Project Manager)

Approver

David Cooper (Technical Authority)

Copies to:

NSA

Praxis High Integrity Systems
Project File

Contents

1 Introduction

1.1 Background

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 develop a high integrity variant of an existing secure system (the Tokeneer System) in accordance with their own high-integrity development process. The component of the Tokeneer System that is to be redeveloped is the core functionality of the Token ID Station (TIS). This development work will then be used to show the security community that it is possible to develop secure systems rigorously in a cost-effective manner.

1.2 Purpose

This document describes the verification process applied to the software and summarises the results of the static analysis and formal code verification that was performed.

1.3 Structure

The next section describes in general the process of performing Formal verification using SPARK.

Section 3 describes the verification process applied to the TIS code and the properties of the TIS code that were formally verified.

Section 4 summarises the results of the verification process.

2 Code verification using the SPARK Examiner toolset

The SPARK Examiner provides a number of facilities for performing static analysis of the code. This static analysis is achieved by inserting specification information into the source code in the form of annotations. Some of the annotations are mandatory and the code is not analysable without these, other annotations are optional and allow more detailed levels of analysis. The SPARK Examiner then analyses the code against the annotations. There are a number of levels of analysis that can be performed by the SPARK Examiner, these are covered in the following sections and summarised in Figure 1.

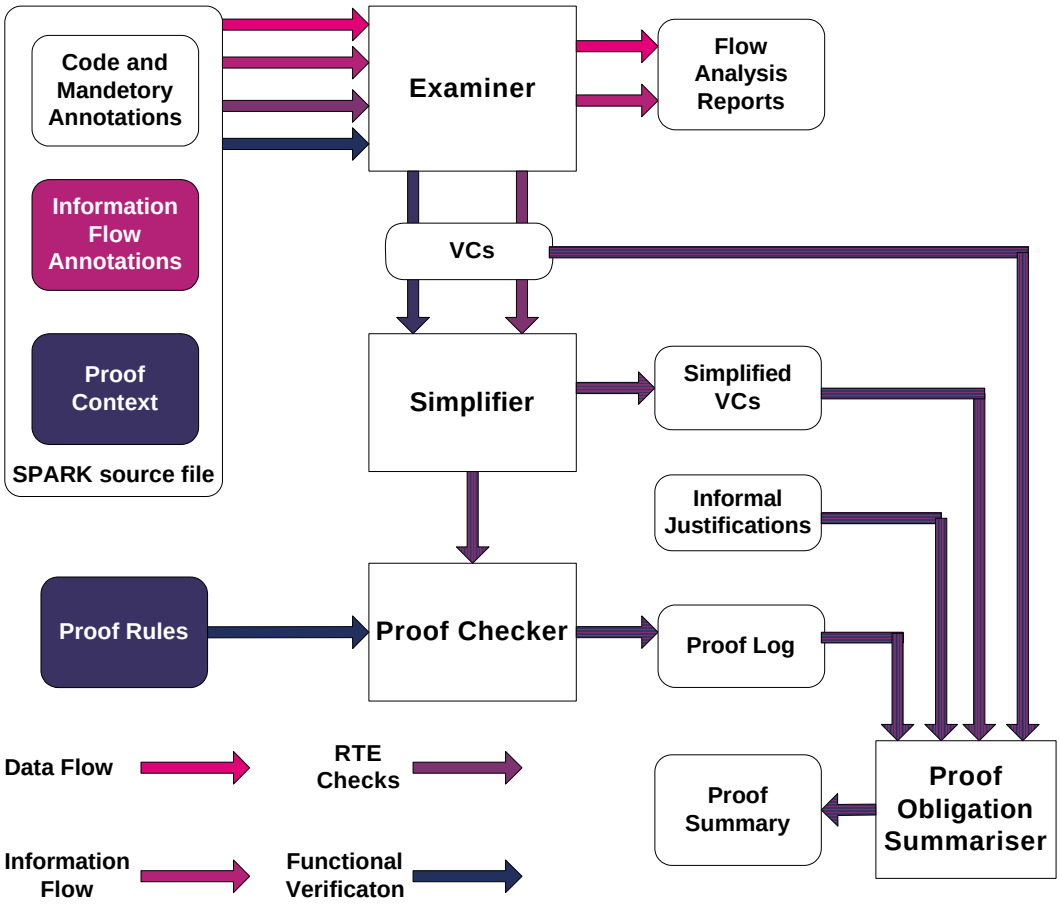


Figure 1 The SPARK Analysis options

Round cornered boxes represent inputs or outputs while square cornered boxes represent tools in the SPARK Examiner Toolset.

2.1 Syntactic and Semantic Analysis

The SPARK Examiner performs this by default. The SPARK language is a subset of Ada conforming to a number of checkable constraints along with mandatory annotations that make explicit all the global

state that must be in the scope of each package or subprogram. These constraints ensure that the code is statically analysable; they also eliminate potentially erroneous uses of the Ada language.

2.2 Data Flow Analysis

The minimal level of analysis also includes data-flow analysis, for this to be performed **global** annotations need to be present. Global annotations specify all the global state that may be used or modified by a subprogram. The SPARK Examiner checks the consistency of these annotations with the code and checks for key data flow properties such as initialisation before use of state, at both the local and global level.

2.3 Information Flow Analysis

With the addition of **derives** annotations the SPARK Examiner can additionally perform information flow analysis. Derives annotations specify the expected interdependencies between the global state components in the system. The SPARK Examiner checks the consistency of these annotations (the specified dependencies) with the code. Information flow analysis is optional.

2.4 Run-time Error Checking

Run-time error checking is an optional, but very powerful, check that may be performed by the SPARK Examiner with just the minimal global annotations present and the addition of assertions relating to type information of variables used in loops. Run-time error checking analyses the conditions that must be satisfied to ensure that no exceptions are raised at each point in the code where an Ada run-time exception may be raised. The SPARK Examiner generates its results in the form of a number of VCs (Verification Conditions), all of which must be satisfied to guarantee freedom from run time exceptions. With the use of the SPADE Simplifier these VCs can be reduced to a small number which need to be proved True, the final proof effort can either be performed manually using rigorous argument or by using the Proof Checker, an interactive proof tool.

Software that is proven run-time error free can be guaranteed never to overrun array bounds or set variables to values outside their type. These can be very important properties since they are mechanisms by which malicious attacks can be made on code.

2.5 Functional Verification

By annotating subprograms with proof contexts, that take the form of **pre** and **post** conditions it is possible to prove functional properties of the code. The level of functional proof achieved depends on the detail provided within the proof contexts, these can describe the full functional behaviour of a subprogram or just key properties. As for proof of absence of run-time errors, the SPARK Examiner generates VCs that must be proved true to demonstrate that the code does indeed meet its specified post conditions, within the context of its preconditions.

Functional Proof can also be assisted by the definition of proof functions and provision of rewrite rules for these proof functions. The rewrite rules can be applied in order to discharge VCs using the Proof Checker.

2.6 Proof Summary

The Proof Obligation Summariser summarises the results of all proof activities. This indicates the source of all VCs generated (RTE checks, precondition checks etc) It also summarises the point in the Proof process in which the VC was discharged (shown true). This is a very useful mechanism for ensuring that all obligations have been discharged.

3 Verification of TIS Core software

All of the TIS Core software has undergone the following checks:

- Data-Flow Analysis
- Information-Flow Analysis
- Run-time Error Checking

The software implementation followed the structure of the Formal Design [1]. This allowed **global** and **derives** annotations to be deduced from the scope of the Z schemas being implemented and the formal relationship between state components as defined by the schema predicates. See Appendix A for examples that demonstrate the correspondence between the Z Design and the flow annotations.

SPARK annotations were included at an early stage in the coding, prior to the implementation of subprogram bodies. The SPARK Examiner was run early and often during code development to ensure that each subprogram that was developed conformed to the specified flow properties.

Once all the code in a package was complete run-time error checking was performed on the code, this was done before code review and the SPARK output was part of the material reviewed.

As a consequence of the early and regular application of static analysis and run-time error checking many errors were detected and corrected by the developer as part of the implementation phase before the code even reached review. This is a phenomenon that has been exhibited on a number of projects where SPARK Analysis has been applied early, in contrast to projects where static analysis has been applied retrospectively, after the code has been completed.

3.1 Functional Verification

As the code resembled the Z Design very closely (see Appendix A for evidence) it was decided that it would not be cost effective to perform full functional proof. The reason being that the similarity between the code and the Z was sufficiently transparent that it was effectively and easily checked by code review.

Functional verification was used to prove certain key invariants (corresponding to state invariants in the Z design) were maintained by the code.

We also demonstrated that a sample of the security properties were preserved by the code. Preservation of the security properties is less obvious from reading the code, but by inserting pre and post conditions we were able to re-express the security properties within the SPARK annotations and then prove that the code did exhibit these properties.

The security properties demonstrated were:

- A partial demonstration of Security Property 1 which denotes the conditions under which the door may be unlocked by TIS. In particular we focused on the Guard having performed an Override Lock operation.

- A full demonstration of Security Property 3, which requires that the alarm be raised whenever the door is in an insecure state.

Time rather than technical limitations prevented us from proving that all the security properties were preserved by the code.

3.1.1 Verification of invariant on audit log

We proved a key invariant that defined the number of audit entries in terms of the total number of used audit files and the size of the current audit file. Within the Formal Design there is an invariant defining *numberLogEntries* in the state schema *AuditLogC.*

$$usedLogFiles \neq \emptyset$$
$$\quad \wedge\ currentLogFile = last\ usedLogFiles$$
$$\quad \wedge\ numberLogEntries = (\#usedLogFiles - 1) * maxLogFileEntries + \#(logFiles\ currentLogFile)$$
$$\vee$$
$$usedLogFiles = \emptyset \wedge numberLogEntries = 0$$

Formally the invariant is stated as follows in the SPARK proof context – note that within the code the number of used log files is never zero.

```
--# pre NumberLogEntries = LogEntryCountT(UsedLogFiles.Length -1)*MaxLogFileEntries +
--#                        LogFileEntries(CurrentLogFile);
--# post NumberLogEntries = LogEntryCountT(UsedLogFiles.Length -1)*MaxLogFileEntries +
--#                        LogFileEntries(CurrentLogFile);
```

This invariant was shown to hold for all public subprograms that modify the values of `UsedLogFiles`, `NumberLogEntries` or `CurrentLogFile` within the audit log package. From this we can be sure that the invariant is always maintained by operations accessed by the public interface.

We also show that the initialisation procedure `Init` establishes this invariant.

Proving this invariant holds of the code helped establish the correctness of one of the most algorithmically complex aspects of the code.

3.1.2 Verification of Security Property 3

Security Property 3, taken from [3], is as follows:

An alarm will be raised whenever the door/latch is insecure.

"insecure" is defined to mean the latch is locked, the door is open, and too much time has passed since the last explicit request to lock the latch.

There are two places in which real world updates occur:

$$Update \mathrel{\widehat=} TISEarlyUpdate \lor TISUpdate$$

$$\begin{array}{l} Update \mid \\ \quad latch' = locked \\ \quad \land\ currentDoor' = open \\ \quad \land\ currentTime' \geq alarmTimeout \\ \vdash \\ \quad alarm' = alarming \end{array}$$

This is clearly a property of the whole system so we would expect it to hold of the TIS main program. We reformulate this property in the SPARK proof context as follows:

```
--#   ( ( Latch.IsLocked(Latch.State) and
--#       Door.TheCurrentDoor(Door.State) = Door.Open and
--#       Clock.GreaterThanOrEqual(Clock.TheCurrentTime(Clock.CurrentTime),
--#                                Door.prf_alarmTimeout(Door.State)) )
--#            ->
--#       Alarm.prf_isAlarming(Alarm.Output) )
```

However, in proving this property we determined that the code does not implement this exactly. This is because there is a possibility that a peripheral has failed, so we cannot determine its state. In this case the code raises a critical SystemFault (not formally modelled in the Formal Design). So what we can prove of the code is

```
--#   ( ( Latch.IsLocked(Latch.State) and
--#       Door.TheCurrentDoor(Door.State) = Door.Open and
--#       Clock.GreaterThanOrEqual(Clock.TheCurrentTime(Clock.CurrentTime),
--#                                Door.prf_alarmTimeout(Door.State)) )
--#            ->
--#       Alarm.prf_isAlarming(Alarm.Output) or SystemFault )
```

Considering the security property in more detail we see that this has been is written in terms of SPARK functions, which are present in the code, such as `Door.IsLocked` and SPARK proof functions, these are functions declared within annotations which are only present within the Proof context, by convention the names of proof functions are always prefixed with `prf_`. In the above post condition we observe two proof functions. `Alarm.prf_isAlarming` and `Door.prf_alarmTimeout`, these retrieve information from the packages Alarm and Door that is not made visible within the code. `Alarm.prf_isAlarming(Alarm.Output)` returns true exactly when the current alarm output is alarming (not silent).

By use of the SPARK Toolset we have proved that the code satisfies this property.

To do this the above property was introduced as a post condition to the `MainLoopBody` procedure that implements the main loop activities of TIS.

```
   --# post
   --#      ( ( Latch.IsLocked(Latch.State) and
   --#          Door.TheCurrentDoor(Door.State) = Door.Open and
   --#          Clock.GreaterThanOrEqual(Clock.TheCurrentTime(Clock.CurrentTime),
   --#                                   Door.prf_alarmTimeout(Door.State)) ) ->
   --#        ( Alarm.prf_isAlarming(Alarm.Output) or SystemFault ) ) ;
is
begin
   Poll.Activity(SystemFault => SystemFault);

   if not SystemFault then
      Updates.EarlyActivity(SystemFault => SystemFault);

      if not SystemFault then

         Processing;

         Updates.Activity(SystemFault => SystemFault,
                          TheStats    => TheStats,
                          TheAdmin    => TheAdmin);

      end if;
   end if;
end MainLoopBody;
```

Then pre and post conditions were applied to each of the subprograms used by the `MainLoopBody` that modify any of the state components referenced in the security property. Any procedure that modifies some or all of this state can potentially break the property. By appropriate choice of pre and post conditions we decompose the problem of proving the security property into smaller proof obligations on smaller code fragments. This divide and conquer approach is very powerful in decomposing the proof obligation into smaller obligations on smaller code fragments which can be proved using the available tool support.

The proof contexts for the procedures called within the `MainLoopBody` procedure are as follows:

```
--# post
--#      ( ( Latch.IsLocked(Latch.State) and
--#          Door.TheCurrentDoor(Door.State) = Door.Open and
--#          Clock.GreaterThanOrEqual(Clock.TheCurrentTime(Clock.CurrentTime),
--#                                   Door.prf_alarmTimeout(Door.State)) ) <->
--#        Door.TheDoorAlarm(Door.State) = AlarmTypes.Alarming )
--#
```

Figure 2 Proof context for Poll.Activity relevant to Security Property 3

Polling does not impact the Alarm.Output, rather the Door alarm state records the requirement for the alarm to be raised (during the Update phase).

```
--# post
--#      ( Door.TheDoorAlarm(Door.State) = AlarmTypes.Alarming ->
--#          Alarm.prf_isAlarming(Alarm.Output) )
--#
```

Figure 3 Proof context for Update.EarlyActivity and Update.Activity relevant to Security Property 3

Following a poll it is sufficient for the alarm to be raised whenever the Door alarm state records the requirement for the alarm to be raised.

```
--# pre  ( ( Latch.IsLocked(Latch.State) and
--#          Door.TheCurrentDoor(Door.State) = Door.Open and
--#          Clock.GreaterThanOrEqual(Clock.TheCurrentTime(Clock.CurrentTime),
--#                                   Door.prf_alarmTimeout(Door.State)) ) <->
--#        Door.TheDoorAlarm(Door.State) = AlarmTypes.Alarming ) ;
--# post ( ( Latch.IsLocked(Latch.State) and
--#          Door.TheCurrentDoor(Door.State) = Door.Open and
--#          Clock.GreaterThanOrEqual(Clock.TheCurrentTime(Clock.CurrentTime),
--#                                   Door.prf_alarmTimeout(Door.State)) ) <->
--#        Door.TheDoorAlarm(Door.State) = AlarmTypes.Alarming ) ;
```

Figure 4 Proof context for Processing relevant to Security Property 3

This states that any changes made by the processing activity preserve the property attained by the Poll.Activity.

These proof contexts are sufficient to prove that the `MainLoopBody` does indeed satisfy Security Property 3. So the problem is reduced to showing that each of the procedures called within the `MainLoopBody` satisfies its proof contexts.

The whole decomposition of the proof reduces to demonstrating that the operations `Poll`, `LockDoor` and `UnlockDoor` in the Door package satisfy the proof context given above for `Poll.Activity`. Analysis of these subprograms shows that they all invoke the local procedure `UpdateDoorAlarm`, which sets the Door alarm state according to the conditions stated in the proof context in Figure 5

```
--# post
--#      ( CurrentDoor = Open and
--#        Latch.IsLocked(Latch.State) and
--#        Clock.GreaterThanOrEqual(Clock.TheCurrentTime(Clock.CurrentTime),
--#                                 AlarmTimeout) ) <->
--#      DoorAlarm = AlarmTypes.Alarming;
--#
```

Figure 5 Proof context UpdateDoorAlarm relevant to Security Property 3

The proof context here replaces all state from the Door package that was previously retrieved by functions and proof functions, by the state used in the implementation.

The SPARK Toolset (Examiner and Simplifier) is able to discharge all proof obligations relating to `UpdateDoorAlarm` – demonstrating that the implementation satisfies its proof context. The equivalence between this proof context and that used elsewhere in the software is proved by demonstrating that the proof functions and retrieval functions do indeed retrieve the stated state components. So for example it is necessary to demonstrate that `TheCurrentDoor` function returns the value of `CurrentDoor`.

The SPARK Toolset does analysis of the whole calling tree, ensuring the security property is maintained for every path through the code.

3.1.3 Verification of Security Property 1

Security Property 1, taken from [3], is as follows:

If the latch is unlocked by the TIS, then the TIS must be in possession of either a User Token or an Admin Token. The User Token must either have a valid Authorisation Certificate, or must have valid ID, Privilege, and I&A Certificates, together with a template that allowed the TIS to successfully validate the user's fingerprint. Or, if the User Token does not meet this, the Admin Token must have a valid Authorisation Certificate, with role of "guard".

$$
\begin{array}{l}
\Delta IDStation;\ \Delta RealWorld \mid \\
\quad TISOpThenUpdate \\
\quad \wedge\ latch = locked \wedge latch' = unlocked \\
\vdash \\
\quad (\exists\ ValidToken \bullet goodT(@ValidToken) = currentUserToken \\
\qquad \wedge\ UserTokenOKNoCurrencyCheck \\
\qquad \wedge\ FingerOK) \\
\quad \vee \\
\quad (\exists\ TokenWithValidAuth \bullet goodT(@TokenWithValidAuth) = currentUserToken \\
\qquad \wedge\ UserTokenWithOKAuthCertNoCurrencyCheck) \\
\quad \vee \\
\quad (\exists\ ValidToken \bullet goodT(@ValidToken) = currentAdminToken \\
\qquad \wedge\ authCert \neq \emptyset \wedge (the\ authCert).role = guard)
\end{array}
$$

Again this is a property that holds of the system as a whole so it results in a proof obligation on the TIS main program. We translate this proof obligation to the SPARK proof language to give the following proof context.

```
--#      ( ( ( Latch.prf_isLocked(Latch.Output~) and
--# ( ( ( Latch.prf_isLocked(prf_preLatchOutput) and
--#       not Latch.prf_isLocked(Latch.Output) and
--#       Latch.IsLocked(prf_preLatchState) = Latch.prf_isLocked(prf_preLatchOutput)
--#     )
--#     ->
--#     ( UserEntry.prf_UserEntryUnlockDoor or
--#       ( AdminToken.prf_isGood(AdminToken.State) and
--#         AdminToken.prf_authCertValid(AdminToken.State) and
--#         AdminToken.TheAuthCertRole(AdminToken.State) = PrivTypes.Guard )
--#     )
--#   )
--#   or SystemFault )
```

Figure 6 Proof Context for Security Property 1 in TIS main program

Here prf_preLatchOutput and prf_preLatchState refer to the initial state of the external latch(Latch.Output) and internal view of the latch state (Latch.State) respectively. prf_isGood and prf_authCertValid characterise properties of the certificates on the Admin Token.

Again taking into account the possibility of a system fault occurring, either a System Fault occurs or the internal latch state matches the real latch and the security property holds, in that if the latch becomes unlocked then either user entry has resulted in the door being unlocked (characterised by `prf_UserEntryUnlockDoor`) corresponding to the first two disjuncts in the Z statement of the security property. Alternatively a guard is present with a valid authorisation certificate.

We only demonstrated the security property was maintained with regard to the Guard being present (and having overridden the lock). The remainder of the proof relating to user entry was not progressed further – this could have been done, had time permitted, by elaborating the proof function `prf_UserEntryUnlockDoor`.

The proof technique for discharging this security property was similar to that for security property 3, however there were a number of system invariants required to achieve the proof. This is because validation and logging on an administrator is a multi-phase process controlled by the enclave status, as the validation progresses we can assume properties of the admin token and these properties need to be interpreted as system invariants to allow the security property to be successfully discharged.

The required system invariants are as follows:

1 If a guard is present then the Admin Token is Good, with a valid authorisation certificate and the role indicated by that certificate is “guard”.

```
--#      ( Admin.prf_rolePresent(TheAdmin) = PrivTypes.Guard ->
--#         ( AdminToken.prf_isGood(AdminToken.State) and
--#           AdminToken.prf_authCertValid(AdminToken.State) and
--#           AdminToken.TheAuthCertRole(AdminToken.State) = PrivTypes.Guard ))
```

2 If an administrator is performing the Override Lock operation then the administrator must be a guard.

```
--#      ( ( Admin.IsDoingOp(TheAdmin) and
--#          Admin.TheCurrentOp(TheAdmin) = Admin.OverrideLock ) ->
--#           Admin.prf_rolePresent(TheAdmin) = PrivTypes.Guard )
```

3 If a guard is present then either they are performing the Override lock operation or they are not performing an operation.

```
 --#      ( Admin.prf_rolePresent(TheAdmin) = PrivTypes.Guard ->
--#           ( ( Admin.IsDoingOp(TheAdmin) and
--#               Admin.TheCurrentOp(TheAdmin) = Admin.OverrideLock ) or
--#             not Admin.IsDoingOp(TheAdmin) ))
```

4 If there is no administrator present then no administrative operation is in progress.

```
--#      ( not Admin.IsPresent(TheAdmin) -> not Admin.IsDoingOp(TheAdmin) )
```

5 If the enclave status is either gotAdminToken or waitingRemoveAdminTokenFail then there is no administrator present.

```
--#      ( ( Enclave.prf_statusIsGotAdminToken(Enclave.State) or
--#          Enclave.prf_statusIsWaitingRemoveAdminTokenFail(Enclave.State) ) ->
--#        not Admin.IsPresent(TheAdmin) )
```

6 If the enclave status is either waitingStartAdminOp or waitingFinishAdminOp then an administrator is present and doing an operation.

```
--#      ( ( Enclave.prf_statusIsWaitingStartAdminOp(Enclave.State) or
--#          Enclave.prf_statusIsWaitingFinishAdminOp(Enclave.State) ) ->
--#        ( Admin.IsDoingOp(TheAdmin) and
--#          Admin.IsPresent(TheAdmin) ) )
```

7 If the enclave status is enclaveQuiescent then there is no administrative operation in progress.

```
--#      ( Enclave.prf_statusIsEnclaveQuiescent(Enclave.State) ->
--#        ( not Admin.IsDoingOp(TheAdmin) ) )
```

8 If the enclave status is shutdown then there is no administrator present and no administrative operation in progress.

```
--#      ( Enclave.prf_statusIsShutdown(Enclave.State) ->
--#        ( not Admin.IsDoingOp(TheAdmin) and
--#          not Admin.IsPresent(TheAdmin) ) )
```

9 If enrolment is in progress then there is no administrator present and no administrative operation is in progress.

```
--#      ( Enclave.EnrolmentIsInProgress(Enclave.State) ->
--#        ( not Admin.IsPresent(TheAdmin) and
--#          not Admin.IsDoingOp(TheAdmin) ) )
```

These invariants are sufficient to allow proof of the security property with regard to the administrator overriding the lock. It should be noted, and can be proved, that the only administrator operation that can cause the latch to become unlocked is the Override Lock operation.

The code was proven to satisfy the portion of security property 1 relating to the guard overriding the lock. This was achieved using the SPARK Toolset.

4 Results of the verification process

The Proof Obligation Summariser provides a facility to summarise the results of all proof activities, this includes proof of absence of run-time errors and functional verification.

In general we only used the SPARK Examiner and Simplifier in our proof activity. We did however use the Proof Checker to prove the outstanding VCs within the `TisMain` package and the `Enclave` package. In other packages where there were un-discharged VCs we used (manual) informal justification to discharge the outstanding VCs.

The summarised results are presented in Figure 7. This summary categorises the source of each VC as well as indicating where the VC was discharged.

The results for TIS are typical of a system developed using SPARK from a formal specification. Either the Examiner or the Simplifier automatically discharged 95.8% of the VCs, leaving just 4.2% to be verified manually. We used a combination of review and use of the Proof Checker to discharge the remaining VCs.

```
Overall subprogram summary:
------------------------------------------------------------------------
Total subprograms fully proved:                                    285
Total subprograms with at least one undischarged VC:                 1  <<<
Total subprograms with at least one false VC:                        0
                                                                   -----
Total subprograms for which VCs have been generated:               286

VC summary:
------------------------------------------------------------------------
Note: U/R denotes where the Simplifier has proved VCs using one or more user-
defined proof rules.

Total VCs by type:
                          -----------Proved By Or Using------------
                    Total  Examiner Simp(U/R)  Checker Review False Undiscgd
Assert or Post:      1006    561    376             29     40      0       0
Precondition check:    67      0     60              3      4      0       0
Check statement:        1      0      1              0      0      0       0
Runtime check:       1337      0   1331              2      4      0       0
Refinement VCs:       212    182      2              9     19      0       0
Inheritance VCs:        0      0      0              0      0      0       0
========================================================================
Totals:              2623    743   1770             43     67      0       0
% Totals:                     28%    67%             2%     3%     0%      0%
```

Figure 7 POGS Summary for TIS core software

A Appendix: Design to Code transformation

This appendix demonstrates, by example, the way in which the code is developed from the Formal Design. The only functionality that was not modelled formally within the design was the possibility of failures of components outside of the TIS Core. This includes the receipt of error codes back from entities such as the Crypto Library as well as failures of the more physical devices connected to the TIS Core. Within the INFORMED Design [2] the action on receipt of such failures was defined as reporting a *SystemFault* within the audit log. The severity of this audit entry depended on whether the peripheral that had failed had a direct impact on the ability to preserve the security properties or not.

Within the SPARK annotations the possibility of system faults being raised is characterised by procedures updating the audit log when the Formal Design does not show an explicit update.

Taking this into account we see through the following code fragments from the TIS Core software how the annotations are derived from the Formal Design and how a close correspondence has been maintained between the code and the Formal Design. This aids validation of the code and ensures the maintainability of the code.

The following examples consider fragments of code from relatively low-level packages, Door and Cert and a fragment of code from the UserEntry package that implements the top-level user entry operation.

From the Door package we have selected the UnlockDoor operation, which implements the partial operation schema *UnlockDoorC*.

From the Cert package we have selected the IsOK operation, which implements the schema predicate *CertOKC*

From the UserEntry package we have selected the ReadFinger operation, which implements the partial operation schema *TISReadFingerC.*

Formal Design

SetUnlockDoorTimeouts

Δ*DoorLatchAlarmC*
ConfigC

$currentLatchC' = currentLatchC$
$currentTimeC' = currentTimeC$
$latchTimeoutC' = currentTimeC + latchUnlockDurationC$
$alarmTimeoutC' = currentTimeC + latchUnlockDurationC + alarmSilentDurationC$
$currentDoorC' = currentDoorC$
$doorAlarmC' = doorAlarmC$

▷ See: *DoorLatchAlarmC* (p. 34), *ConfigC* (p. 26)

▷ *latchUnlockDurationC* and *alarmSilentDurationC* are imported from *ConfigC*.

Once the timeouts have been reset the latch and alarm must be updated.

UnlockDoorC $\widehat{=}$ *SetUnlockDoorTimeouts* ⨾ *UpdateInternalLatch* ⨾ *UpdateInternalAlarm*

Implementation

```
procedure UnlockDoor
is
   LatchTimeout : Clock.TimeT;
begin

   LatchTimeout := Clock.AddDuration(
                   TheTime     => Clock.TheCurrentTime,
                   TheDuration => ConfigData.TheLatchUnlockDuration
                   );

   Latch.SetTimeout(Time => LatchTimeout);

   AlarmTimeout := Clock.AddDuration(
                    TheTime     => LatchTimeout,
                    TheDuration => ConfigData.TheAlarmSilentDuration
                    );

   Latch.UpdateInternalLatch;
   UpdateDoorAlarm;

end UnlockDoor;
```

Figure 8 UnlockDoor operation from Door package

A demonstration of the close correlation between the formal design and the code. Note that the unchanged state components in the Z design do not get set in the code.

Formal Design

FD.Certificate.SignedOK
FS.Certificate.Validate

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.

CertIssuerKnownC
KeyStoreC
CertificateContents
keyMatchingIssuer idC, issuerC.id ≠ nil

▷ See: *KeyStoreC* (p. 32), *CertificateContents* (p. 16)

A certificate must have been signed by the issuer.

CertOKC
CertIssuerKnownC
RawCertificate
(mechanism, digest mechanism data, signature)
isVerifiedBy (the (keyMatchingIssuer idC, issuerC.id))

Implementation (package specification)

```
-- Traceunit : C.Cert.IsOK
-- Traceto   : FD.Certificate.SignedOK
-------------------------------------------------------------
procedure IsOK (RawCert    : in CertTypes.RawCertificateT;
                Contents   : in     ContentsT;
                IsVerified :    out Boolean);
--# global in     KeyStore.Store;
--#        in     Clock.Now;
--#        in     ConfigData.State;
--#        in out AuditLog.FileState;
--#        in out AuditLog.State;
--# derives AuditLog.FileState,
--#         AuditLog.State     from Contents,
--#             KeyStore.Store, AuditLog.FileState
--#                                  AuditLog.State,
--#                                  Clock.Now,
--#                                  ConfigData.State,
--#                                  RawCert &
--#         IsVerified         from Contents,
--#                                  KeyStore.Store,
--#                                  RawCert;
```

Figure 9 IsOK operation from Cert package specification

A demonstration of the relationship between the Z state and the variables appearing within the derives and global annotations. Other than changes to the audit log, the only variable that may be updated is the isVerified Boolean that corresponds to whether or not the CertOKC schema is satisfied. Notice that IsVerified depends on exactly those state components that appear in the predicate schema CertOKC.

Formal Design

CertIssuerKnownC
KeyStoreC
CertificateContents
keyMatchingIssuer idC, issuerC.id ≠ nil

▷ See: *KeyStoreC* (p. 32), *CertificateContents* (p. 16)

A certificate must have been signed by the issuer.

CertOKC
CertIssuerKnownC
RawCertificate
(mechanism, digest mechanism data, signature)
isVerifiedBy (the (keyMatchingIssuer idC.issuerC.id))

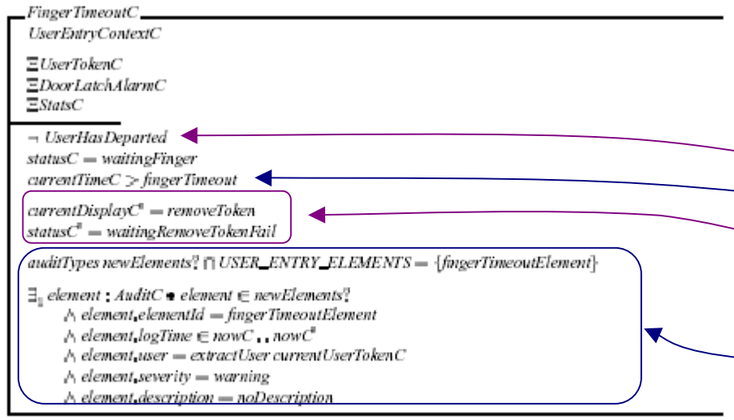
Implementation

```
procedure IsOK (RawCert    : in
CertTypes.RawCertificateT;
                Contents   : in     ContentsT;
                IsVerified :    out Boolean)
is
   IsKnown : Boolean;
begin

   IssuerKnown(Contents => Contents,
               IsKnown  => IsKnown);

   if IsKnown then
      KeyStore.IsVerifiedBy
        ( Mechanism   => Contents.Mechanism,
          RawCertData => GetData(RawCert),
          Signature   => GetSignature(RawCert),
          TheIssuer   => Contents.ID.Issuer,
          Verified    => IsVerified);

   else
      IsVerified := False;
   end if;

end IsOK;
```

Figure 10 IsOK operation from Cert package body

A demonstration of the close correlation between the formal design and the implementation. Here the Boolean IsVerified is set True exactly when CertOKC holds.

Formal Design



Implementation (procedure body)

```
is
  FingerPresence : BasicTypes.PresenceT;
begin

   if not UserToken.IsPresent then

      UserTokenTorn(TheStats => TheStats);

   else
      if Clock.GreaterThan(Clock.TheCurrentTime, FingerTimeout) then
         -- FingerTimeoutC actions

         Display.SetValue (Msg => Display.RemoveToken);
         Status := WaitingRemoveTokenFail;

         AuditLog.AddElementToLog
           ( ElementID   => AuditTypes.FingerTimeout,
             Severity    => AuditTypes.Warning,
             User        => UserToken.ExtractUser,
             Description => AuditTypes.NoDescription);

      else
         Bio.Poll(FingerPresent => FingerPresence);

         if FingerPresence = BasicTypes.Present then
            -- ReadFingerOKC actions

            Display.SetValue (Msg => Display.Wait);
            Status := GotFinger;

            AuditLog.AddElementToLog
              ( ElementID   => AuditTypes.FingerDetected,
                Severity    => AuditTypes.Information,
                User        => UserToken.ExtractUser,
                Description => AuditTypes.NoDescription);

         else
            -- NoFingerC actions
            null;
         end if;
      end if;
   end if;
end ReadFinger;
```

Figure 11 The implementation of the ReadFinger operation explained

Even within the high level UserEntry package there is a clear relationship between the Formal Design and the code which makes verification of the code a trivial exercise. The only line of code that cannot be mapped trivially to the Z schemas above is the Poll of the biometric device. The INFORMED Design tells us that polling of the finger is removed from the upfront Poll activity and postponed to the point where the data is required.

Implementation (procedure specification)	Commentary
`procedure ReadFinger(TheStats : in out Stats.T)`	This is the implementation of the following fragment from the Formal Design
`--# global in ConfigData.State;`	$TISReadFingerC \mathrel{\widehat=} [ReadFingerOKC \lor FingerTimeoutC \lor NoFingerC \lor [UserTokenTornC \mid statusC = waitingFinger]] \setminus (newElements?)$
`--#  in Clock.Now;`	Considering the **global** annotations we observe that:
`--#  in Clock.CurrentTime;`	`UserToken.State` will be modified if the *UserTokenTornC* disjunct holds (where the token is Cleared).
`--#  in FingerTimeout;`	`Status` and `Display.State`, corresponding to *StatusC and currentDisplayC* in the Design, will be changed in all disjuncts other than *NoFingerC.*
`--#  in Bio.Input;`	`AuditLog.State` and `AuditLog.FileState` encapsulate the AuditLogC state and will be modified whenever an audit entry is made, again this applies in all disjuncts other than *NoFingerC.*
`--#  in out Status;`	`Clock.CurrentTime`, `FingerTimeout` and `Bio.Input` correspond to *currentTimeC, fingerTimeout* and *fingerPresenceC* in the Formal Design, are unaffected but are used to determine which of the disjuncts is valid.
`--#  in out UserToken.State;`	The remaining global inputs `Clock.Now` and `Config.State` are used during *`AddElementsToLog`* to time-stamp log entries and determine whether the *auditAlarm* should be raised respectively
`--#  in out Display.State;`	The *currentDisplayC* and *statusC* depend on whether the Token has been torn, whether the *Fingertimeout* has elapsed and whether the finger is present.
`--#  in out AuditLog.State;`	
`--#  in out AuditLog.FileState;`	The *UserTokenC* and the *StatsC* are only modified if the Token has been torn.
`--# derives Status,`	
`--#  Display.State from *,`	
`--#  UserToken.State,`	
`--#  Clock.CurrentTime,`	
`--#  FingerTimeout,`	
`--#  Bio.Input &`	
`--#  UserToken.State,`	
`--#  TheStats from *,`	
`--#  UserToken.State &`	
`--#  AuditLog.State,`	
`--#  AuditLog.FileState from UserToken.State,`	The values of the audit elements added to the log will depend on which of the disjuncts is chosen hence the dependencies on `UserToken.State, FingerTimeout, Clock.CurrentTime`, and `Bio.Input`. Any changes to the display are audited so there is a dependency on `Display.State`. `Clock.Now` is used to time-stamp entries and `Config.State` determines whether the audit alarm is raised (the occurrence of which is itself logged).
`--#  Display.State,`	
`--#  AuditLog.State,`	
`--#  AuditLog.FileState,`	
`--#  ConfigData.State,`	
`--#  Clock.Now,`	
`--#  Clock.CurrentTime,`	
`--#  FingerTimeout,`	
`--#  Bio.Input;`	
`--# pre Status = WaitingFinger;`	The precondition on this procedure reflects the precondition on the *TISReadFingerC* schema.

Figure 12 The specification of the ReadFinger operation explained

For details of the schemas used to construct TISReadFingerC refer to [1].

Document Control and References

Changes history

Issue 0.1 (11 September 2003): Initial Version.

Issue 1.0 (23 September 2003): Issued at Provisional following internal review.

Issue 1.1 (19 August 2008): Updated for public release.

Issue 1.2 (25 August 2008): Updated Figure 7 to show verification results using release 7.6 of the SPARK toolset.

Changes forecast

None.

Document references

1 TIS Formal Design, S.P1229.50.1.

2 TIS INFORMED Design, S.P1229.50.2.

3 TIS Security Properties, S.P1229.40.4