Engineering the Tokeneer Enclave Protection Software

Janet Barnes, Rod Chapman: Praxis High Integrity Systems

Randy Johnson, James Widmaier: National Security Agency

David Cooper: River River Limited

Bill Everett: SPRE Inc.

Publication notes

Published in ISSSE '06, the proceedings of the 1st IEEE International Symposium on Secure Software Engineering, March 2006.

Engineering the Tokeneer Enclave Protection Software

Janet Barnes, Rod Chapman,
Praxis High Integrity Systems
20 Manvers Street,
Bath, BA1 1PX, UK
janet.barnes@praxis-his.com
rod.chapman@praxis-his.com

Randy Johnson,
James Widmaier,
National Security Agency
drjohns@orion.ncsc.mil

David Cooper,
River River Limited, UK.
david.cooper@riverriver.co.uk

Bill Everett,
SPRE Inc,
436 Oppenheimer Drive,
Los Alamos, NM 87544, USA
wwe@spre-inc.com

Abstract

The Tokeneer ID Station (TIS) project, carried out by Praxis High Integrity Systems in conjunction with SPRE Inc. under the direction of NSA, has shown that it is possible to produce high quality, low defect systems conforming to the Common Criteria requirements for Evaluation Assurance Level 5 (EAL5). We state the seven guiding principles we used to achieve this, and relate each one to examples from the TIS development. The systems development industry in general has viewed conformance with the Common Criteria higher levels as too difficult, too expensive, and generally not economical. The experience of Praxis High Integrity Systems, however, is that the levels of EAL5 and beyond (including EAL7) are achievable in a cost-effective manner. This TIS project was commissioned as a demonstration vehicle, to show exactly how the development approach adopted by Praxis matches up to EAL5, and to measure its actual productivity and defect rates under controlled conditions.

1. Introduction

The need for high assurance software, which is correct, complete, reliable, and available goes without saying in the security domain. However, developing high assurance software applications with adequate verification mechanisms has been not only difficult to achieve but the state-of-the-art techniques (often incorporating formal methods) have not been viewed as cost-effective until recently [1]. The semi-formal approach used by Praxis High Integrity Systems using the SPARK [2,3] toolset has successfully moved formal methods tools into the commercial domain where both assurance and cost requirements have been adequately addressed. The cost is now lower than traditional manual object-oriented methods per line of code [4,5] with a measured operational reliability of .9999 (see section 8.1 of this paper and [14]). The complexity of the process has also been managed so that industry can reach capability with cost-effective investments in training and off the shelf tools.

2. Achieving low-defect software

Low-defect software can be achieved by applying a number of techniques, working together. There is no magic bullet technique that solves all the problems—but a coordinated application of a range of techniques does work. We have used some specific techniques (such as SPARK) in this demonstration project, but we have also applied more general guiding principles of development to guide us in appropriate development steps and notations:

Write right. Write in a way that captures the information you want, without confusion, ambiguity, or verbosity. Know what sort of information you want to write, and use notations that support you, not hinder you. Use a programming language with unambiguous static and dynamic semantics.

Step, don't leap. The step between each development phase should be semantically small. Large steps increase the likelihood of error, and make checking for errors harder.

Say something once, why say it again? Each phase of development, and each design specification or document, should have a clearly defined purpose, and express information or involve decisions that have not been expressed or made before. Repetition of the same information in multiple places leads to unnecessary work and can hide the actual design decisions made.

Check here before going there. Each design step should be verified as soon as possible. Make your reviewing effective—review *against* something, usually a prior design representation. So review code against the module specification, the functional specification against the requirements specification, etc. And ensure that validation is easy—write simple code that directly reflects the specification.

Argue your corner. Document the justifications for why design decisions were made, why they are appropriate, and how you ensured that they were carried out correctly. Justifications will help future analysis, but most importantly being forced to document why you do something at the time will help uncover errors at the time.

Screws: use a screwdriver, not a hammer. Use the most appropriate verification technique for the properties you are checking. This may be formal review, informal peer review, tool-supported proof, static analysis, etc.

Brains 'R' Us. Think about everything you do. Carry out careful requirements discussions with your customers, intelligently searching for conflict, write well-structured and clear functional specifications, carefully document well thought-out changes.

3. The Project's Vital Statistics[6]

The system was written in SPARK, a high-integrity subset of Ada with added annotation to allow for design by contract, static analysis and program proof. The core functionality (the true focus of this demonstration project) was written in pure SPARK, and the support software to interface this to simulated peripherals was written in full Ada.

Tables 1 and 2 show vital statistics for the project's size, productivity and effort. The total effort was 260 person-days, with an elapsed time of 1 year using a team of three, all working part-time on the project.

The number of defects in the system found during independent system reliability testing and since delivery is zero[1].

Table 1. Size and productivity

	Size/source lines		Productivity (LOC/day)	
	Ada	SPARK annotations and comments	During coding	Overall
TIS Core	9,939	16,564	203	38
Support Software	3,697	2,240	182	88

Table 2. Effort breakdown by project phase

Project Phase	Effort %	Effort Person-days
Project management	11	28.6
Requirements	10	26.0
System specification	12	31.2
Design core functions	15	39.0
TIS Core code and proof	29	75.4
System test[2]	4	10.4
Support software and integration	16	41.6
Acceptance	3	7.8
Total	**100**	**260.0**

4. The Project

The overall Tokeneer system was originally developed by the NSA as a research vehicle for investigating various aspects of biometrics as applied to access control. The system's overall context is shown in Figure 1. It consists of a secure enclave, physical access to which must be controlled. Within the secure enclave are a number of workstations. Users of the workstations have security tokens (e.g. smartcards) in order to gain entry to the enclave, and to gain access to the workstations. Users present their security tokens to a reader outside the enclave, which uses information on the token to carry out biometric tests (e.g. fingerprint reading) of the User. If the User passes these tests, the door to the enclave is opened and the User is allowed entry. The User also uses their security token to gain access to the workstations—at entry time the Tokeneer system adds authorisation information to the security token describing exactly the sort of access allowed for this visit to the enclave, such as times of working, security clearance, and roles that can be taken on.

[1] The independent testers, SPRE Inc., logged two in-scope failures as a result of their testing: both were aspects missing from the user manual, and hence do not reflect errors in the TIS Core (the part of the system developed to our high-integrity process, and the prime subject of study). See "Independent assessment", below, for the limits to testing.

[2] The proportion of time spent on system testing was extremely low, even compared to other Praxis Correctness by Construction projects. A more representative figure would include the testing contribution from SPRE (the independent testers) including the production of the test environment. A more normal proportion from other Praxis projects is 25%.

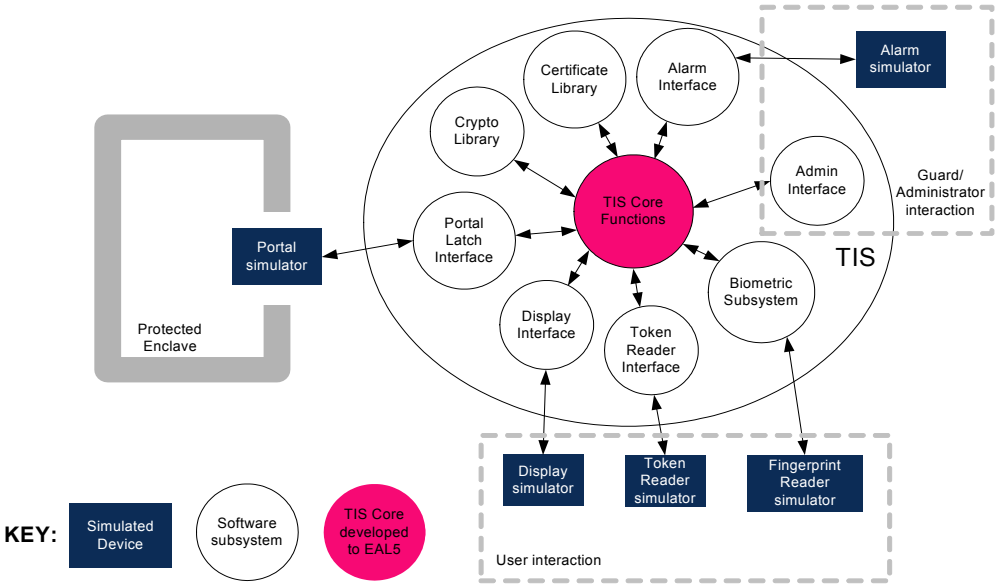

Figure 1. TIS System Context

The Tokeneer ID Station (TIS) project re-developed one component of the Tokeneer system. To facilitate the development, TIS device simulators implemented by the independent reliability testing consultants (SPRE, Inc.) were used in place of actual TIS devices. Communication between the simulators and the kernel was via TCP/IP sockets, allowing for a very flexible development and test environment. Kernel development proceeded in the UK while the device simulators were developed in Albuquerque, NM with the kernel and device simulators communicating over the internet. The core, security-critical functionality was developed according to Praxis's Correctness by Construction approach. The remaining, support software, was developed to good software engineering standards, but not the full Correctness by Construction approach.

5. TIS Kernel Protection Profile

SPRE, Inc. extended the security-related requirements of the existing Tokeneer by first segregating security requirements into a TIS kernel. In essence, the kernel would operate as a controller, managing the devices associated with the TIS.

This approach allowed security assessments and certifications to focus on the kernel. The TIS Kernel Protection Profile (PP) emphasised Operations, Administration, Maintenance and Provisioning capabilities (OAM&P). Since the TIS Kernel would be used in high risk environments, the evaluation assurance level (EAL) was set at 5. The PP is based on Version 2.1 of the Common Criteria [7]. The Common Criteria Toolkit was used in developing the PP.

6. The Development Approach

The development approach adopted on this demonstration project was based on the Praxis Correctness by Construction approach. In outline it follows a conventional sequence, shown in Figure 2, but in its details it is quite radical.

Each step is discussed below, with emphasis on how Correctness by Construction achieves its low defect rate through the application of the seven principles.

6.1. Requirements analysis

The most expensive mistakes to fix are those that occur near the beginning of the development. But they are also easy mistakes to make. Correctness by Construction tackles these issues through a whole requirements engineering approach called REVEAL®, developed by Praxis to ensure that the right tools get used at the right time, and the key information and decisions are documented clearly and unambiguously (use a screwdriver; Write right). An example is the identification of the *system boundary*. This was carried out on the first day of the project, and enabled us to make a clear separation between work being done on the core functionality (and hence would be developed to EAL5 criteria), work being done on supporting software (such as the simulators), and work outside the scope of the project. This also clarified the dependencies we had on aspects of the environment, such as certificates (supplied

®REVEAL is a UK registered trademark of Praxis High Integrity Systems Limited.

by the Certificate Authority) and the behaviour of the door/latch. 48% of requirements failures are due to misunderstandings or changes in the *environment*, not the system, and our requirements engineering techniques reflect the importance of this fact [8].

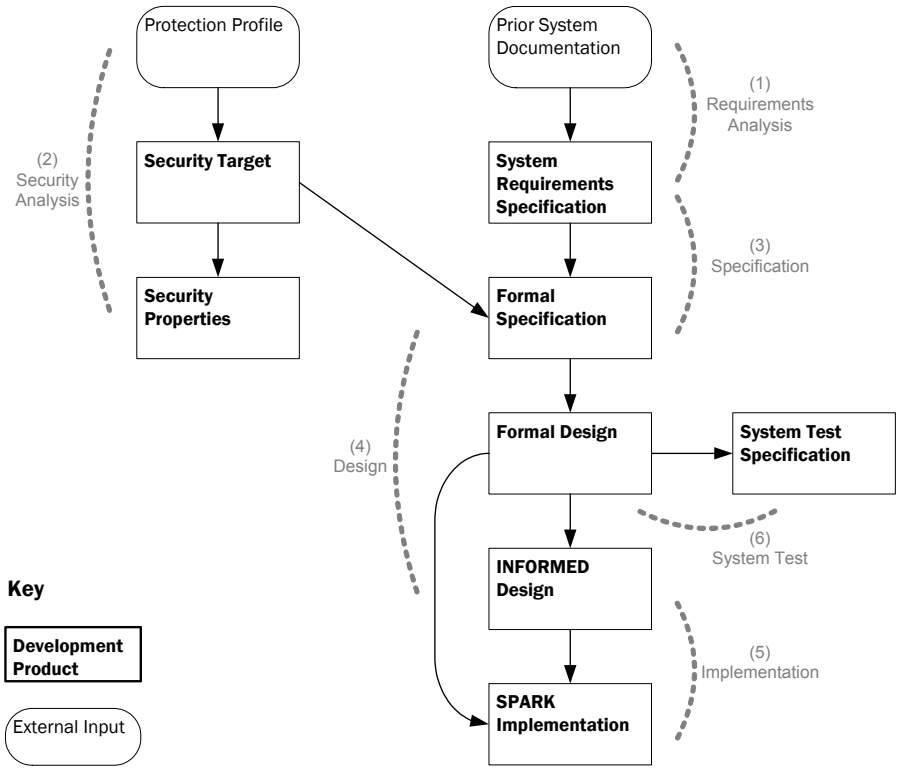

Figure 2. Development Process

Arrows show dependencies between activities to produce development products.

6.2. Security analysis

As this was a development of a *security critical* system, and had development constraints imposed by the Common Criteria, we developed a Security Target and a Security Policy Model, derived from the Protection Profile. These are orthogonal to the normal development of functional specifications, design and code, because they seek to identify only the key properties that must be upheld to ensure security, rather than the functionality needed to make it useful, user-friendly, etc. (Say something once; Write right). This ensures that we really do understand what the properties are, and also gives us the opportunity to demonstrate rigorously that the system specification does actually possess the properties required of it (Check here before going there). Security policy modelling focuses the mind on understanding the true requirements, and can be used early in the system development as a vehicle for gaining agreement on the system requirements. Proof that the functional specification has the security properties stated also acts as a form of validation of the specification.

6.3. Specification

Most development processes agree that specifications and designs should be developed before coding starts. Why? We expect you will agree with the following reasons:

1 If you haven't documented *what* you expect the system to do, the coders will have to decide this for themselves. But the coders are not the customer, and hence may make decisions the customer disagrees with, resulting in a system that does not do what the customer wants.

2 If the coder has nothing to work from, then neither has the tester (how do I know what the correct behaviour is supposed to be?) nor the reviewer (I can see what the code *does* do, but I don't know what it is *supposed* to do).

3 Coders are human, and cannot reliably make large design steps in their heads; they need to make incremental decisions, and have these decisions argued and reviewed.

For these reasons our Correctness by Construction development approach uses functional specifications, design documentation, and test specifications, that capture the necessary information.

We write a rigorous, functional specification using the specification language called Z (Zed) [9,10,11],. Rigorous notations are the only approach that truly allows the system functionality to be written abstractly and unambiguously. (use a screwdriver; Write right). The TIS functional specification, written in Z with accompanying English text, consists of approximately 100 pages. Having completed it, the behaviour of TIS is completely clear, and any disagreements about what TIS does or what TIS depends on in the environment can be discussed objectively in terms of the functional specification. It is like having the system already coded near the beginning of the project (Argue your corner).

By using a notation that allows us to write abstractly, but still unambiguously, we are able to describe the *what* without the *how*. And it frees the developer's brain power. Most of the tough decisions about behaviour, interface dependencies, security trade-offs, etc. can be discussed *and resolved* at the level of the functional specification, without having to struggle in the morass of code (Brains 'R' Us; Step, don't leap).

6.4. Design

In moving from the functional specification to the code we used two further documents as part of this development. (In general, we use more or fewer depending upon the size of the system and the complexity of the design decisions to be made.) We wrote:

- a refinement of the functional specification (again written in Z) (Write right); and
- a guide to the SPARK packages to be developed, their dependencies and their data storage requirements (using notations from INFORMED[12], the design method that supports SPARK) (use a screwdriver; Step, don't leap).

These documents had clear purposes (rather than just being "another step" in designing from requirements to code) (Say something once):

- The refinement documents certain difficult implementation decisions that were made in passing from the functional specification to the code. The key ones were
    - prioritization: choosing which actions would take priority over other actions (where the functional specification left the choice non-deterministic).
    - auditing: defining in detail the structure and content of audit elements, and designing a practical implementation of audit storage to match the abstract behaviour in the functional specification.
    - certificate structure: moving from abstract certificates to realistic, raw streams of bits.
- the documentation from the INFORMED design analysis ensures that the design (as presented in the refinement) makes sense in terms of implementation modules (SPARK packages), and makes visible the flow of information (and hence the degree of coupling) between modules. Linking the design statement with the implementation modules ensures that verification from the design to the code will be straightforward and tool supported. It also captures the precise nature of the abstractions made in linking to the real world from the software system.

6.5. Implementation

The system is coded in SPARK [2,3], an annotated subset of Ada designed for high-integrity system development, which, unlike most other programming languages, has clear, unambiguous semantics. We use SPARK because it *prevents* a large number of common mistakes from being made, such as the use of un-initialized variables, buffer overflows, and incorrect information flow [13]. Static checks by the SPARK toolset can be carried out before code is ready to be run (and design intention can be analysed even before the code is written), involving data and information flow analysis and program proof (Check here before going there; Step, don't leap).

Good coding practice can significantly reduce the introduction of bugs. The INFORMED design process, a design method specifically developed to encourage designs that benefit from SPARK's strengths, will have produced a design with good modularity properties, and the use of SPARK annotations and data/information flow analysis will ensure that these good properties are preserved in the implementation.

6.6. System test

Using SPARK and its static analysis tools eliminates most of the common coding errors that would be picked up by extensive module testing. We develop our code and integrate it into the system producing incremental builds of the total system. In this project, each build of the system was complete, in that it runs without test harnesses, but with progressively more functionality at each build. This tests the most problematic aspects of coding (integration problems) throughout the development, and removes the need for an expensive integration phase (Brains 'R' Us; use a screwdriver).

We test against the refinement of the functional specification. Specification based testing is very much more effective when the specification is rigorous because the specification covers *all* system behaviour, sufficiently clearly that all possible behaviours, correct and error behaviours, are defined. In addition to specification based testing, we also use automated test tools to measure code coverage, and supplement our tests to achieve 100% statement and branch coverage (Check here before going there).

6.7. Assurance

Each step carried out in the development process, and each decision made, has the potential for error. We are only human, after all. But by having a clear purpose for each document, and appropriate notations, it is possible to check the correctness of decisions by analysing the documents. The figure below shows the analysis performed on the various products of the development process (Check here before going there; use a screwdriver; Argue your corner).

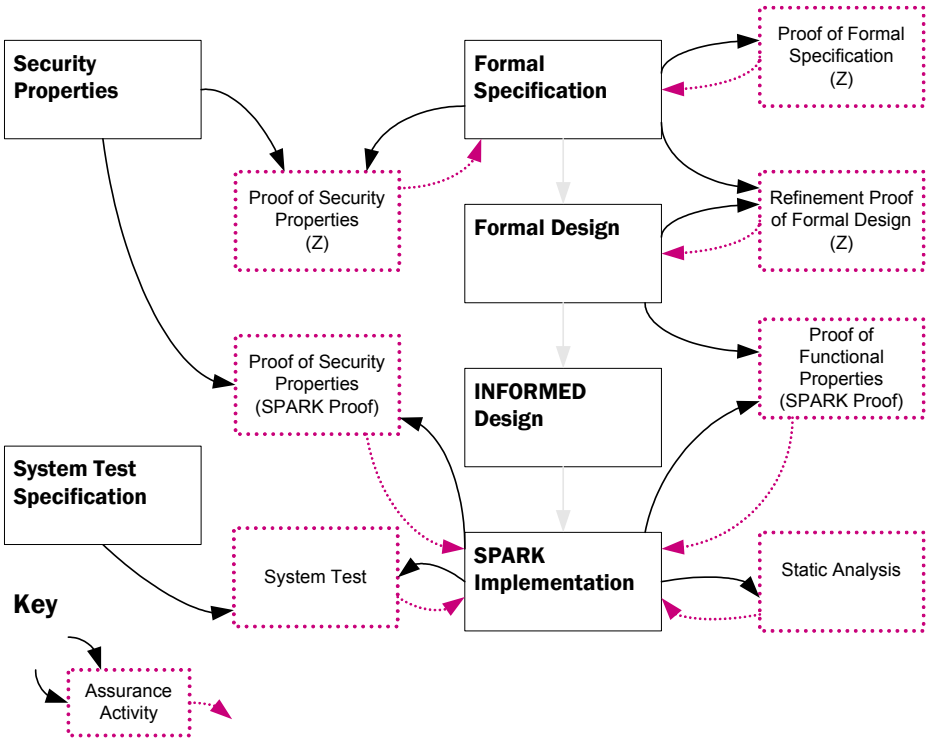

Figure 3. Assurance Process

Arrows into an assurance process indicate the inputs to the assurance process. Arrows out of an assurance process indicate the lifecycle product being validated by the assurance activity.

7. The View from the Customer

Any serious development methodology starts with getting the system requirements right. The approach followed here is no exception. What will be new to many customers is the crucial early jump from the terminology of the problem domain and the English language to concepts expressed in Z, a rigorous notation based on mathematical logic and set theory. The developer is primarily responsible for capturing the specifications in Z, but only the customer, or a trusted and capable representative, can check that this vital step has been achieved correctly. In principle, a single person with full knowledge of the problem domain and a fluent reading knowledge of Z could do this review. We relied on the next best thing, a partnership of a few people with domain knowledge and one person fluent in Z. This approach worked well. We even had one productive review session by phone with the Z expertise in Maryland and the domain knowledge two time zones and thousands of miles away in Albuquerque, New Mexico.

Experience on this project and others has shown that, with good instruction, an adequate reading knowledge of Z can be acquired in a few days. Much depends upon the quality of the accompanying English text. The explanatory English from Praxis on this project was

excellent. Even so, it is always good for the customer to plan for quick access to someone fluent in Z to help translate any difficult idioms, which may appear in documents presented for review. The ability to write professional quality Z takes more than a few days to learn, but customers would not expect to have to do that.

Because of the relatively small size of this project, we had only one level of design document between the specification in Z and the annotated code in SPARK. Reviewing this design document against the specification was straightforward. Not only were both documents written in Z with English explanatory text, but the organization was very similar. Diagrams of system states and transitions were also supplied to help the reader. It was not difficult to accomplish a design review using one person whose background included Z, general system design principles, and the scope and function of this system.

There was no need for a detailed customer review of the SPARK. It was evident that the organization of the implementation was, once again, very similar to that of the design and even a very rudimentary reading knowledge of SPARK was enough to see that the translation was quite direct.

Because this was a methodology assessment exercise with a limited budget, we did not do all of the detailed evaluation that the Common Criteria would call for at the highest assurance levels. To do their job properly, evaluators would clearly need a good reading knowledge of Z with access to someone fluent in Z for help with the tricky bits. A good course of a few days duration and some additional practice could give them this level of Z expertise. They would also need a reading knowledge of SPARK and enough familiarity with the SPARK tools to do at least some spot-checking of proofs. This exercise provides no estimate of the effort required to gain the needed expertise.

8. Independent Assessment

As well as producing the TIS Kernel Protection Profile and implementing the TIS Device Simulators, SPRE carried out independent certification of the reliability of the TIS Core.

8.1 Reliability Certification of the Core

SPRE adapted the Reliability Requirements for the overall Tokeneer to the specifics of the TIS Core. These identified three failure severity categories (critical, major, minor), provided descriptions of types of failure for each category and spelled out specific failure rate objectives for each category. The reliability requirements also specified the operating conditions under which the objectives were to be met. Eight Operational Modes were identified, and Operational Profiles specifying the operations that occur and their frequency of occurrence were given for each mode:

1 Normal Day
2 Installation/Upgrade
3 Configuration Changes
4 Uninstall/Zeroization
5 Backup/Restore
6 Power Outage/Recovery
7 Audit File Review
8 “Outlier” Scenarios

The “outlier” scenarios are those situations that occur very infrequently but in which failures could be critical, e.g., allowing admittance to the enclave during the transition from “daylight” to “standard” time. Software Reliability Engineered Testing [14] methods and Reliability Demonstration Charts were used to certify that the system objectives were met.

Within an automated testing environment, a test schedule was setup to simulate one year’s worth of activity, executing test cases with frequencies specified by the operational profile. We were able to achieve a high degree of automation in executing test cases which allowed us to simulate a year of TIS activity in a couple of weeks of test activity. A failure taxonomy was establish to categorize whether failures were associated with hardware, software or the test-environment. Application software failures were further categorized as to whether they were “in scope” or “out of scope” for the expectations of this demonstration. A review board reviewed reported failures to ensure they were correctly categorized. Figure 4 is a Reliability Demonstration Chart analysing failure incident data, choosing an operational reliability target against *all* in-scope errors of .9999.

The chart was created using the following parameters:

Discrimination ratio = 2:	the value against which the failure intensity is compared. In this case, we are comparing against a failure intensity twice as stringent as the objective.
Supplier risk = 10%:	the likelihood that we erroneously *reject* the product against its reliability objective.
Consumer risk = 10%:	the likelihood that we erroneously *accept* the product against its reliability objective.

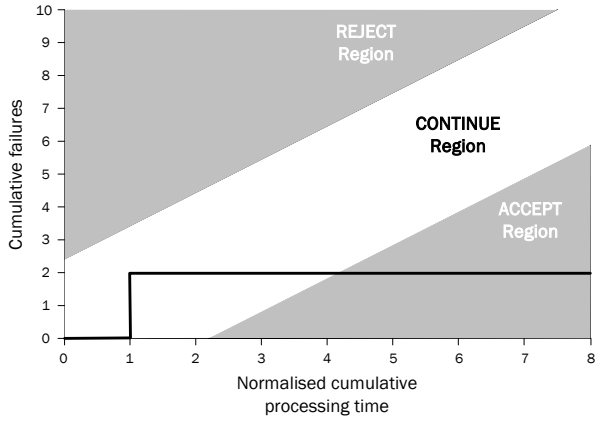

Figure 4. Reliability Demonstration Chart (Minor Failures) *Severity 3 (minor) failures are plotted against the time at which they are observed. There were no severity 1 (critical) or severity 2 (major) failures.*

Failures are plotted against the time at which they were observed. Testing continues until the line falls within the Accept or Reject Regions. At that point, we can say at a specified level of confidence that the system has met or not met its reliability objectives. The chart shows that we can accept that the reliability objectives for the TIS kernel have been met at a 90% confidence level (i.e., there is only an 10% supplier or consumer risk). Although we observed additional failures, these were deemed out of scope for the development project being investigated.

Of course, we can never be *sure* that the TIS Core contains zero defects—that would require exhaustive testing, which is completely impractical. The independent reliability testing shows that, for the operational scenarios defined, the reliability exceeded .9999 (and we have a 90% statistical confidence that this conclusion is true).

9. Exceeding EAL5

The task we were set by the NSA was to develop a system in conformance with the requirements laid out in the Common Criteria for EAL5 [7] (within the scope of the system agreed). In fact, we exceeded the EAL5 requirements in a number of areas, because our Correctness by Construction approach has shown that it is actually more cost-effective to use some of the more rigorous techniques.

We met the EAL5 criteria in the main body of the core development work, covering configuration control, fault management, testing. We exceeded EAL5, coming up to EAL6 or EAL7 levels, in the development areas covering the specifications, design, implementation and demonstration of correspondence between representations. Some aspects were out of scope, such as delivery and operational support.

10. Conclusions

The TIS development project has demonstrated that the Praxis Correctness by Construction development process is capable to producing a high quality, low defect system in a cost effective manner following a process that conforms to the Common Criteria EAL5 requirements.

The TIS system's key statistics are:

- lines of code: 9939
- total effort (days): 260
- productivity (lines of code per day, *overall*): 38
- defects found post delivery per 1000 lines of code: zero

The development approach applied on this project, and described in this report, is Praxis High Integrity System's standard high-integrity development process, and has been applied successfully to a number of commercial and government projects by Praxis. It is not new or under development—it is a proven approach using proven technology. It has been shown to work on information processing systems, interactive systems, and real-time systems [15,16,17]. Our experience in working with other system developers is that our development approach can be applied successfully by other companies, but the learning curve for many organizations is steep. Good training, a continuing mentor and coaching programme, and commitment to improvement are necessary to ensure that take-up of the approach is successful.

This case study has shown that software-based security products can be built that are reliable, verifiable, and cost effective against Common Criteria guidelines. The bar has been raised for both procurers and suppliers of such systems. Two questions remain: is the process scalable to very large systems and can it be followed by anyone other than the professionals at Praxis? We cannot do a research exercise to answer the first question: large real-world system development requires a large real-world budget. We researchers never have such a budget. Evidence of scalability must be found in Praxis's reports of their experience [3,15,16,17].

On the other hand, we can carry out an exercise to see how well people unfamiliar with the methods and technologies involved can successfully use this approach. In fact, we have already done such an experiment with student interns as a test population. But that's another story for another time.

11. References

[1] National Cyber Security Partnership, *Security Across the Software Development Life Cycle ,* http://www.cyberpartnership.org/init-soft.html

[2] John Barnes, *High Integrity Software: The SPARK Approach to Safety and Security,* Addison Wesley, April 2003. ISBN 0-321-13616-0.

[3] SPARKAda homepage, http://www.sparkada.com/

[4] Robert Musson, "How the TSP impacts the Top-Line" *CrossTalk Journal*, Volume 15, Number 9, September 2002. http://www.stsc.hill.af.mil/crosstalk/2002/09/

[5] Smidts, Huang, and Widmaier, *"*Producing Reliable Software: An Experiment"*, Journal of Systems and Software*, August 2002.

[6] Janet Barnes and David Cooper, "EAL5 Demonstrator: Summary Report*"*, Praxis High Integrity Systems, S.P1229.81.1, 17 December 2003.

[7] *ISO 15408, Common Criteria for Information Technology Security Evaluation*, August 1999 (Version 2.1).

[8] Hooks and Farry. *Customer Centred Products*, Amacom, 2000.

[9] J.M. Spivey. *The Z Notation: A Reference Manual*, 2nd Edition, Prentice-Hall, 1985.

[10] *The Z Notation*, http://vl.zuser.org/

[11] *Information technology – Z formal specification notation – Syntax, type system and semantics, ISO/IEC 13568:2002*, International Organization for Standardization, July 2002 .

[12] *INFORMED Design Method for SPARK*, Praxis High Integrity Systems, S.P0468.42.4, January 2005

[13] Dorothy E. Denning, Peter J. Denning, *"*Certification of Programs for Secure Information Flow*", CACM* Vol. 20, No. 7, July 1977, pp504-513.

[14] Musa, John D and James Widmaier, "Software-Reliability-Engineered Testing*", CrossTalk Journal*, Volume 9, Number 6, Software Technology Support Center, June 1996. http://www.stsc.hill.af.mil/crosstalk/1996/06/

[15] Anthony Hall and Roderick Chapman, *"*Correctness by Construction: Developing a Commercial Secure System*"*, *IEEE Software*, Jan/Feb 2002, pp18-25.

[16] Anthony Hall, *"*Using Formal Methods to Develop an ATC Information System*"*, *IEEE Software*, March 1996, pp66-76.

[17] Steve King, Jonathan Hammond, Rod Chapman and Andy Prior, *"*Is Proof More Cost-Effective Than Testing?*"*, *IEEE Transactions on Software Engineering*, Vol 26 No 8, pp675-686, August 2000.