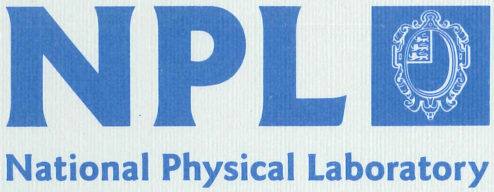
NPL Report DITC 217/93



Testability of data security standards

Roger Manning, Bronia Szczygiel and Andrew Harry

February 1993

The National Physical Laboratory is an Executive Agency of the Department of Trade and Industry

NPL Report DITC 217/93

February 1993

NATIONAL PHYSICAL LABORATORY

Testability of data security standards

Roger Manning, Bronia Szczygiel and Andrew Harry

Division of Information Technology and Computing

Abstract

The report examines the requirements of security evaluation. It investigates areas that must be addressed in defining and testing implementations of standards, including such problems as inadequate specification of security, programming language and compiler insecurities, and difficulties in testing embedded systems. For each of these issues the report suggests solutions and/or areas of research likely to address these problems.

ISSN 0262-5369

National Physical Laboratory
Teddington, Middlesex, UK, TW11 0LW

Approved on behalf of Chief Executive, NPL,
by Mr A J Marks, Head, Division of Information Technology and Computing

Contents

Introduction 1
1 Problems with current methodologies 1
2 Defining security evaluation 2
3 Formal methods 2
3.1 Problems with programming proofs in security 3
3.2 Suggested remedy 5
3.2.1 Conformance clauses 5
3.2.2 Security clauses 5
3.2.3 Expanding formal methods 5
4 Programming languages 6
4.1 Insecurities in programming languages 6
4.2 Suggested remedy 7
4.2.1 The ITSEC and programming languages 8
5 Compiler security 8
5.1 Insecurities in compilation 8
5.2 Suggested remedy 8
5.2.1 Validated Compilers 8
5.2.2 Reverse engineering 9
6 Embedded systems 9
6.1 Types of embedded systems 11
6.2 Problems with testing embedded systems 11
6.3 Suggested remedy 12
6.3.1 Testing inaccessible embedded systems 12
6.3.2 Testing modules in an embedded system 15
6.3.3 Trusted paths 16
Conclusions 16
Acknowledgements 17
References 18
Glossary 21
Appendix 1: Further reading 23

Introduction

As systems become more complex the need for adequate security grows. Computer security is becoming essential to the social infrastructure of our society, particularly in the areas of privacy of the individual (for example hospital records) and finance (banking).

Like any defensive system it is necessary to constantly take in new developments and to assess their impact on the field in question. Defence can be compared to the red queens race in *Alice through the looking glass*[1] where it was necessary to constantly keep moving in order to just stay still. In the area of computing, where there is always rapid development this is particularly so; the move towards the networking of large systems and increasing compatibility between computer bases increases the chances of security leakage and the incentives to subvert the system.

The NPL Data Security Group has looked at ways of formalising the description and testing of data security standards and their implementation in a coherent methodology [24] and this will be discussed within this report. The test methodology must be comprehensive; this is particularly important in the security area where a security breach might only become conspicuous in unique situations. There must also be consistency between test results so that any evaluation is reproducible between different test laboratories.

The report looks at areas of specification and testing applicable to security evaluation, examining the problems encountered in each area and from this drawing conclusions for solutions or future research.

Implementations of data security standards are likely to be embedded in larger systems (for example an OSI[2] protocol), the report therefore also examines the difficulties of testing implementations which are embedded within other systems, discussing problems and possible solutions in this area.

1 Problems with current methodologies

Currently most IT standards are written informally in some form of natural language and, whilst this encourages readability, it can lead to ambiguities and inconsistencies which are difficult to detect. Lack of formalism leads to a test methodology that is not clearly defined and rather haphazard, where implementations of the same standard may be subject to different tests even when being evaluated to the same level (for example, in the OSI field, evaluation test cases are supplied by the sponsor and consequently may vary from implementation to implementation). This results in a lack of certainty about what the evaluation is trying to achieve.

A standardised methodology of specification and testing is necessary to ensure a more general understanding of what is required, leading to tests that are repeatable and reproducible allowing

---

[1] A famous British childrens book.

[2] Open Systems Interconnection: A model for communications which aims at achieving the interconnection of different computers. Developed with the International Standards Organisation (ISO).

comparability of results and a mutual recognition of test results. This in turn should lead to standardised test suites and testing tools.

2 Defining security evaluation

As stated in the previous section it is often unclear what security evaluation entails. We need a clear concise statement of the properties we wish to evaluate.

To be secure it must be ensured that an implementation of a particular standard does *only* what the standard says and does not contain any covert code which will reduce the integrity of the system. In regard to this the Data Security Group defined the term *Strict Conformance Testing*.

Strict Conformance Testing (SCT) involves the testing and analysis of an implementation of an IT standard to ensure that:

(i) it implements the mandatory requirements of the standard in a correct manner,

(ii) it implements those optional requirements of the standard stated as being supported in the conformance statement and tests that they are implemented in a correct manner,

(iii) it contains no functionality which would either be prejudicial to the correct operation of the implementation or would cause a possible breach of security.

NB A conformance statement would be supplied by the implementer, detailing which options of the standard have been implemented.

SCT is the necessary minimum requirement for properly testing security-critical software, testing which is less rigorous is not likely to be able to guarantee its suitability for such applications. SCT is a general representation of the properties we wish to look at in a security evaluation; we would wish to convert these requirements into specific security properties for each security standard.

3 Formal methods

Formal methods offer one way of making the specification and testing of software more rigorous. Based on an underlying principle of logic and set theory they enable a more thorough way of specifying software standards, not susceptible to many of the deficiencies encountered with natural language specifications [11].

The Data Security Group's work into the use of formal methods in the writing of standards and testing of implementations of these standards involved looking at how security properties can be defined within a formal framework and how standardised conformance testing techniques might be created to evaluate these properties utilizing the advantages that formal methods give.

With this in mind the Data Security Group has produced reports on the use of formal methods within the context of the design and implementation of secure software [6,20,21,24], constructing formal descriptions of the peer entity authentication algorithm [22], the MD4 message digest

algorithm [12,13] and the Message Authenticator Algorithm (MAA) [19,26,28,29] in a number of formal languages.

These algorithms are all concerned with some security aspect of protocols. The peer entity authentication algorithm is a two-way handshake protocol concerned with the process of mutual authentication between users or processes on a network. The MD4 and MAA are both data authentication techniques employed in the production of a unique signature or fingerprint from a message, which can be used by the recipient of the message to ensure it has not been tampered with.

The formal specification produced for the MAA was used to create an implementation [23]; this helped highlight some of the advantages and limitations of current formal methods in the area of security specification and analysis.

3.1 Problems with programming proofs in security

When producing an implementation from a specification we attempt to produce an equivalent piece of code in a different semantic base; going from the specification semantics to that of the programming language by a process called translation.

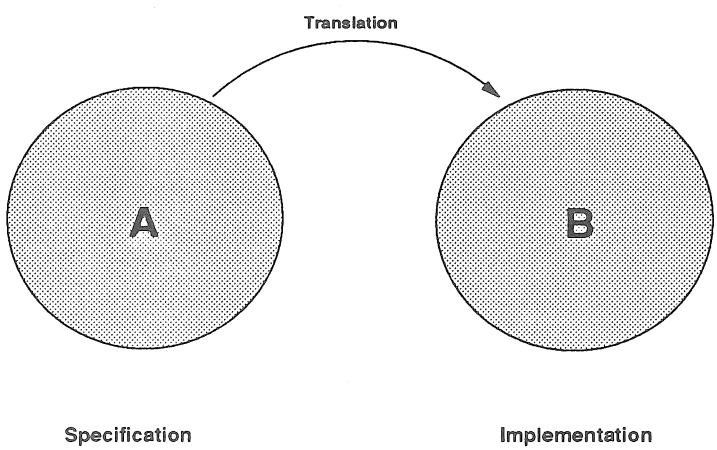


Fig 1: Refinement and translation (ideal view)

Fig 1 represents this in its ideal form. Unfortunately this does not take into account the fact that the implementation has to deal with the real world, whereas the specification, being of an abstract mathematical nature, does not. Extra code is usually required to deal with real world situations such as user interfaces, exception handlers and error messages; these are by their very nature unverifiable and it has been estimated that more than half the code in real production systems consists of this sort of code [9].

Limitations of real world facilities also represent a problem. Whilst it is quite reasonable to use the infinite set of natural numbers in an abstract specification, this is not feasible on a real system and a finite limit has to be given on the size of a representation.

These problems result in an implementation that is far from the ideal model given in Fig 1 and the result is more likely to be a mélange of complex programs that have informally derived aspects embedded within them (Fig 2).

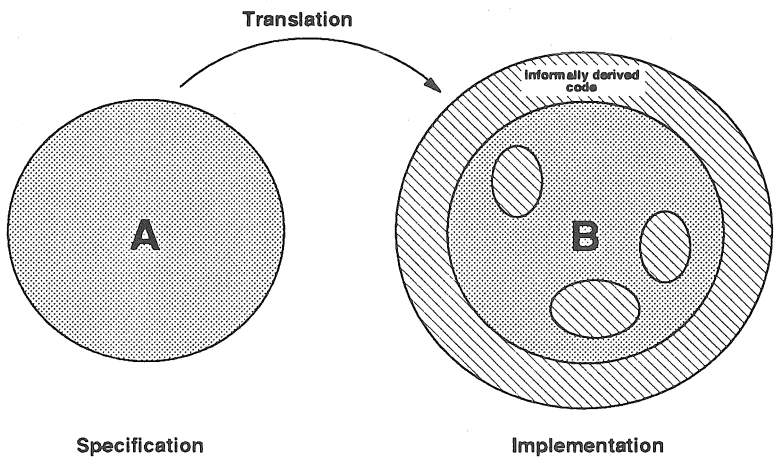


Fig 2: Refinement and translation (practical view)

This means that legitimate code, not specified in the standard, may be included in the implementation to cover such things as interfacing and exception handling. It is therefore necessary to distinguish between this type of code and illegitimate code which may have been placed in the program to subvert the security of the system.

Normal formal proof analysis methods seek only to verify that a program carries out those details given in the specification. It does not check for additional extraneous functionality, except in the context of where it affects the required output of a specification (the postcondition). This severely limits the value of proof analysis in the area of security evaluation.

Stated in set theory, formal proof analysis seeks to show (where Spec is the specification and Imp the implementation) that the implementation does everything specified in the specification:

$$Spec - Imp = \{\}$$

In the security context, we also wish to additionally prove that the implementation does no more than the specification:

$$Imp - Spec = \{\}$$

Except, we have to allow additional 'legitimate' code to cope with certain real world problems.

$$Imp - Spec = \{legitimate\ code\}$$

Since this 'legitimate' code has not been formally specified, there is currently no way of differentiating it from 'illegitimate' code, except by manual perusal.

This problem exacerbates the difficulty of using proof analysis in a security context. With present states of technology it is unlikely that any practical proof analysis method could be carried out without a great deal of manual inspection and interpretation of the implementation. On a medium to (very) large system this form of manual inspection would plainly be impractical, very labour intensive and could not guarantee the security of a system (since there exists the possibility of overlooking details); ways to aid this type of inspection are therefore necessary.

3.2 Suggested remedy

As we have seen in their present state, formal methods do not cover all areas of code in an implementation so they cannot in themselves, as yet, form a comprehensive test structure but would require supplementary information. The Data Security Group has looked at ways of supplementing the formal description to allow comprehensive testing by the use of conformance and security clauses (see below) in standards. It is also examining techniques capable of expanding formal descriptions enabling them to describe security properties such as confidentiality.

3.2.1 Conformance clauses

In OSI the use of conformance clauses plays a vital role in supporting the conformance testing methodology, providing an explicit statement for each OSI standard of what constitutes a conforming implementation. In OSI this can be stated in the form that the external behaviour of the implementation is consistent with having implemented the protocol specification detailed in other specified clauses of the standard. This includes specifying which features are mandatory and which are optional.

The introduction of such clauses to data security standards is seen as an important step in the establishment of a testing methodology.

3.2.2 Security clauses

Similar to a conformance clause, a security clause addresses the security-related properties that an implementation of a standard must demonstrate. In particular it examines the problems resulting from the relationship of software to its environment; problems in this area are expanded in sections 3 and 4 of this report. The rationale for security clauses is discussed in [30].

3.2.3 Expanding formal methods

Although formal methods are valuable in allowing the rigorous definition of security standards, as we have shown we are currently unable to define all the properties we wish to encompass in security specification and evaluation. Formal security policy models [41] allow us to define some security properties in a formal way, but these are very limited in the areas they can represent. Work has, however, been carried out by the Programming Research Group in Oxford into a

formal way of defining confidentiality properties [14,15] and the suitability of this to specifying security properties needs to be examined.

4 Programming languages

The programming language in which an algorithm is implemented can have a direct affect on its security. Insecurities have been found in a number of languages leading to the conclusion that the language chosen may be of importance when accessing the security of a system.

4.1 Insecurities in programming languages

If an implementation produces operational errors during its execution but there were no programming or compiler errors, then these execution errors are likely to have been caused by insecurities present in the programming language. The errors are caused by deficiencies in the descriptions of the programming language and are unlikely to be detected by a compiler. Insecurities of this type, of special concern in the safety-critical area, exist for example, in the programming language Ada[3] and are described in [34].

Another type of insecurity is of special importance in the security-critical domain, where a programming language may have inherent weaknesses which affect the security of an implementation, which a programmer may be unaware of. A resulting implementation may therefore be operationally correct (ie contain no errors) but is nevertheless vulnerable to attack, even though the programmer had taken what were thought to be adequate safeguards in maintaining security. Insecurities of this type (also in Ada) are described in [36]. There is however, some debate about the importance of these findings, as discussed in [38].

Much of the work in investigating weaknesses in programming languages has been carried out in the safety-critical area and a report [7] compares assembly-level languages, and the high level languages C, CORAL 66, Pascal, Modula-2 and Ada. The following extract from the report concludes that software designers should consider in descending order of merit, the following languages:-

(i) ISO Pascal subsets supported by validation tools (e.g. SPADE Pascal);

(ii) an Ada sub-language, (e.g. SPARK Ada kernel);

(iii) a Modula-2 sub-language, when available;

(iv) a CORAL 66 subset;

In spite of potential insecurities, Ada is widely used in applications including both safety and security-critical aspects and also for programming embedded systems [2].

---

[3] It should be noted that Ada is in fact better than most languages in dealing with these issues and should not be regarded as the main offender in this regard.

4.2 Suggested remedy

A recent defence standard on safety-critical systems [33] in considering implementation languages, stated:

> Any programming language used for the implementation of SCS (Safety Critical Software) should have a formally-defined syntax. ... In order for the code to behave as expected there needs to be a link between the Formal Design and the execution of statements in the implementation language. This link is provided by a formal semantics for the language, which ideally is shown to be consistent with both the rules for carrying out formal proofs about the code (the proof theory) and the design of the compiler.

It concluded that:

> These requirements are now attainable for languages of modest size; in fact, they have been met by very small languages in research publications for over a decade. But no existing full language (e.g. ISO-Ada) meets it, and the most satisfactory approach at the present time is to define subsets of languages (e.g. Spark Ada). These subsets are based on the full language, but have dangerous but non-essential constructs excised; dangerous constructs are those that may be ambiguous, hard to analyse, or difficult to implement in a compiler; examples are the use of functions and procedures as parameters, and variant records.

There are specific problems to be considered in the security-critical area (as described in [36] for example), and these should be investigated to the same degree as has been done for safety-critical software when choosing a programming language for a security critical application. The use of a rigorously defined languages will also assist in analysis of the code during testing, since many of the test methods suggested for security evaluation [24,42] have difficulty dealing with constructs such as pointers. Others, such as reverse translation [18], rely on a formally defined implementation language and could not be used unless the language has been defined in this way.

There has undoubtedly been much research carried out in the military and national security sectors into investigating security weaknesses in programming languages used in security-critical applications, but less in the commercial sector. There is currently not a great deal of information available to guide an implementer of a security-critical application in the choice of which language or subset to use: it would seem sensible to at least adopt the conclusions as described previously for safety-critical software. Such restrictions on the choice of language may constrain a developer when producing an implementation, but they must be seen as a necessary restriction for some security-critical applications. This form of restriction could be contained in the standard through a security clause (see section 3.2.2). An example of this is given in [30] for the MAA standard where it is stated that:

> The programming language in which the standard is implemented must not weaken the algorithm.

4.2.1 The ITSEC and programming languages

The Information Technology Security Evaluation Criteria (ITSEC) [41], are the harmonised criteria of France, Germany, the Netherlands and the United Kingdom (and now adopted by the European Community) that address the requirements for a programming language as follows (in Assurance Correctness Level E3, Aspect 2 - Programming Languages and Compilers):-

> Any programming languages used for implementations shall be well-defined, e.g. as in an ISO standard.

A well defined language in this context is one whose semantics have been clearly defined.

5 Compiler security

5.1 Insecurities in compilation

Even if there are no security weaknesses in the source code of an implementation, there remains another means of compromising the security of the executable code, which is an attack on the compiler which introduces covert code into the executable code. A report [31] describes a sophisticated Trojan horse attack on a compiler: a means of removing such an attack is described in [27].

5.2 Suggested remedy

Possible solutions in this area are to use validated compilers or reverse engineering.

5.2.1 Security Verified Compilers

A security-verified compiler would be a compiler which has been stamped by a certifying agency, who would inspect it and then certify it as containing no subversive code which could be used to add security weaknesses to an implementation created using it. Once created a security-verified compiler could be placed on a read-only media (such as CD, ROM etc) or stamped [35] to ensure that it could not be altered; it could then be distributed. This would ensure that the compiler could not be altered after certification.

It is often simpler to validate familiar languages if they are restricted to 'secure' subsets. Security-verified compilers could also stipulate a particular subset of a language, disallowing insecure constructs.

The use of a security-verified compilers could be stipulated within the security clause of a standard (section 3.2.2).

5.2.2 Reverse engineering

Reverse engineering [4], describes the process of deriving a high level language representation from its low-level equivalent, for example a source code version from an assembler version. (in the usual process the high level version results in assembler or object code, after compilation). A paper [3] describes a possible method of reverse engineering of object code to a high-level representation of the program.

Used in conjunction with reverse translation [18,25] (where implementation code is translated into formal specification code) it may be possible to convert low level assembly code back into a formal specification which can be compared against the original specification.

Currently reverse engineering is at a very early stage of development, but it is important to be aware of future work in this area as it is a potentially useful technique.

6 Embedded systems

Many implementations will be embedded within a larger system and it may be difficult to carry out security evaluation within the larger environment. The Data Security Group has looked into the particular problems that might be encountered with this form of software.

Fig 3 is a schematic representation of how a data security system consisting of **trusted** (security enforcing and security relevant functions) and **untrusted** (non-security relevant functions) components might be constructed. The trusted software component is area A, which includes the embedded area *C*.

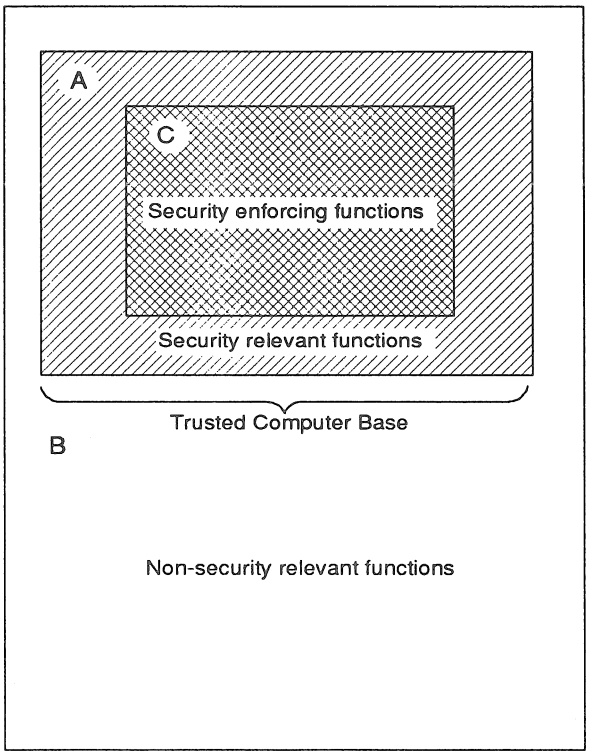


Fig 3: A schematic representation of a data security software system

There are various ways in which a complete system concerned with security might be built, for example, the system may be inside the trusted software component, with the trusted component *A* and *C* completely embedded within the untrusted component. However, different architectures may have similar requirements concerning the use of trusted and untrusted software.

The amount of trusted functionality in the complete system should be as small as possible, consistent with maintaining security and functionality. This reduces the amount of rigorous security-relevant testing required for the trusted component: the untrusted component is not tested with security as a requirement, hence it is tested less rigorously.

As the untrusted component *B* is not concerned with security aspects, any malicious code contained within it is not able to compromise the security of the complete system, given that the boundary between the untrusted and trusted components has been correctly specified and implemented (this is represented by the dividing line between components *A* and *B* in Fig 3).

Such trusted software is said to be enclosed within a **Trusted Computer Base** (TCB), or security perimeter. The embedded systems we are concerned with in security evaluation are the security enforcing components of the system and as such form part of the TCB.

Two examples of criteria which apply to such software are that:

- Software in the untrusted (non-security relevant) component should not be able to access the memory used by the trusted component (TCB).

- Variables and functions in the trusted component (TCB) should be hidden from the untrusted (non-security relevant) component.

A complete system needs to interact with the real world via an interface to be of any practical use. Such an interface could interact with human users and/or another system and is often security relevant, as it could handle secret keys, PINS, or plaintext, for example. Such security relevant software which is not part of the embedded system is represented in Fig 3 by the component *A*. Such interface software however requires the rigorous testing procedures of trusted software, which then adds to the extent of the trusted component.

The system may be tested using various techniques to prove that it behaves according to its specification. However, the specification of the system is unlikely to contain details of how it will be incorporated into a completed system as this would reduce its generality. Additional code may therefore be required to enable it to execute in its operating environment. This extra code, which depends on how the embedded system is intended to interact with the rest of the system, is also contained in the TCB.

As the resulting amount of trusted code could be extensive, there is ample opportunity for it to contain code which could threaten its security, by intercepting security-critical data crossing to and from the embedded system for example. However it is important that *all* software which is security relevant is included in the Trusted Computer Base *A*.

6.1 Types of embedded systems

Embedded systems can be of three types:

(i) The code representing the embedded system is distributed throughout the code of the whole system, making it difficult to differentiate from the rest of the software. It may share the same functions and modules with the rest of the system, making it difficult to identify the embedded component

NB The sharing of library functions, I/O routines, etc which are part of the programming language is to be expected.

(ii) Although the system is self contained, it has been hidden, or encapsulated in some way to prevent access to it. This may be for security and/or commercial reasons, and may be carried out by some form of scrambling, or encryption of the program code.

(iii) The embedded system is of type (i) and has been hidden, or encapsulated in some way to prevent access to it, as in (ii).

Whilst type (i) is difficult to test, types (ii) and (iii) are impossible, unless some means of unscrambling the code is available. It may be, however, that an embedded system does not execute meaningfully unless additional code is added to it. Such considerations are dealt with later.

6.2 Problems with testing embedded systems

Testing an embedded system may be difficult since there is no direct access to it. It is likely to be inextricably linked into a system and indirect access to it through an interface may be complex.

Some of the problems which may be encountered when testing embedded systems in the safety-critical area have been investigated to ascertain if there are any useful techniques which may also be applicable to security-critical testing. Some of these are described in [37], which gives examples of embedded systems carrying out industrial process-control, aircraft flight guidance, hospital patient monitoring, radar tracking, ballistic missile defence, and data acquisition for experimental equipment.

The following edited extract from [37] describes the special problems encountered when testing such embedded systems:

> The special nature of embedded systems exacerbates many software engineering problems, and thus demands particular attention even during the requirements phase. Coping with complexity and change will not be easier in the domain of embedded systems. In addition to the performance requirements, embedded systems are likely to have stringent resource requirements. The interface between an embedded system and its environment tends to be complex, asynchronous, highly parallel and distributed. Embedded systems can be extraordinarily hard to test. The complexity of the system environment interface is one obstacle, and the fact that these programs often cannot be tested in their operational environments is another.

As it was often impossible to test the types of embedded system described in the previous section in the environment in which they would eventually be deployed, the solution, described in [37], was to:

> Specify the requirements for an embedded system with an explicit model of the proposed embedded system interacting with an explicit model of the system's environment. Both sub-models consisted of sets of asynchronously interacting digital processes, although some of the processes in the environment model could represent discrete simulations of non-digital objects such as people or machines. The entire model was executable, and the internal computations of the processes were specified in an applicative specification language.

In this case, an operational model of the system functioning in its environment was constructed. This was carried out using a specially constructed language, PAISLey (Process-oriented, Applicative, and Interpretable Specification Language) which is an approach to the problem of specifying the requirements for embedded systems by offering extra formality, expressive power and potential for automation. The main aim of this approach is to minimise errors at the specification and design phases of the embedded system, although this solution is of course little help to third party testers who have not themselves developed the system.

In the security-critical domain only some of the problems described in the above extracts may be relevant for a given application. Human safety may not be a consideration, but software faults in the security-critical domain could have serious financial effects on individuals, companies, or even nationally.

6.3 Suggested remedy

Apart from the previous solution of creating an operational model using a specially constructed language, the following suggests ways in which embedded systems might be tested:

6.3.1 Testing inaccessible embedded systems

An embedded system may be inaccessible to a tester because it has been purposely designed that way for commercial and/or security reasons. This may have been carried out by some means of encapsulation such as encryption: as it is not the task of the tester to penetrate these encapsulation techniques, sufficient documentation should be provided to the tester to enable access to the embedded system.

Alternatively, the implementation may only be embedded due to its specification, but not physically, within the whole system: this means that the code comprising the embedded system is distributed throughout the whole system. This poses problems to a tester who has to identify the components comprising the embedded system to enable it to be tested. Documentation describing the location of code comprising the embedded system should made available to the tester: an implementation of this type may ultimately prove impossible to test properly as far as security criteria are involved.

NB The ITSEC requires separation of a Target of Evaluation (TOE) into security relevant and non-security relevant functions in levels above E1.

If the embedded system *is* identifiable as a contiguous piece of code separable from the rest of the system, its interactions and interfaces may nevertheless be sufficiently complex that the tester is not able to be confident that it has actually been isolated from the rest of the system. If this is the case the embedded system may either have to be tested as part of the whole system (which may not be feasible), or the tester will have to discover the interactions within the system. It is necessary to demonstrate that the functions outside the TCB cannot interfere with those inside.

Software tools may of assistance here by:-

(i) Producing a list of functions used in the complete system, which may be checked against those which should be used in the embedded component.

(ii) Producing a list of interactions between functions to check whether those used in the embedded component interact with those outside it.

Such tools may also be used to check for violations of the TCB.

A methodology for conformance testing of multi-layer protocol implementations is defined in ISO/IEC 9646 - single layer embedded testing [39]. The Implementation Under Test (IUT) may consist of several layers, each of which has to be tested separately although there may be only one Point of Control and Observation (PCO) through which access is gained. In this case, the layers are tested one at a time with test data directed at the layer under test being accompanied by simple Protocol Data Units (PDUs) for the other layers:

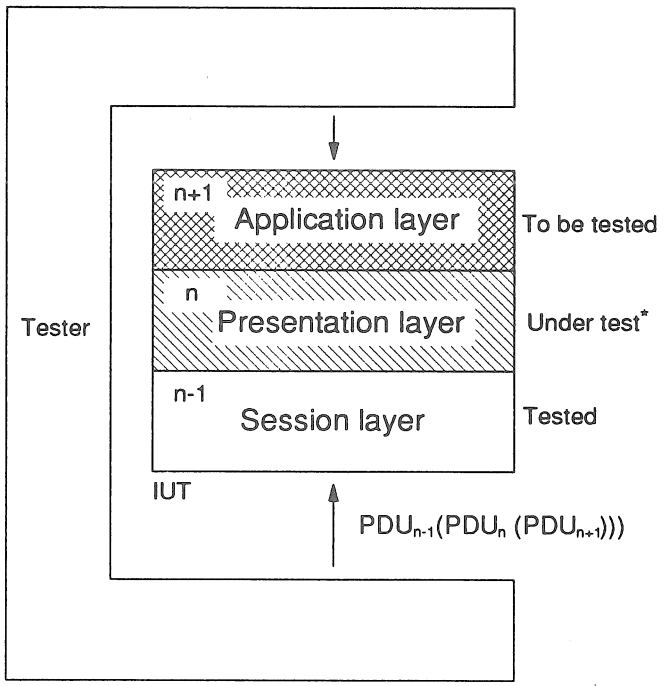


Fig 4 : Test data for $layer_n$ through $layer_{n-1}$ with a $layer_{n+1}$ PDU for testing

Testing would start at the lowest layer of the IUT and progress upwards incrementally. For each layer then the lower layers will have already been tested and will be considered correct. The layers above will be an unknown quantity and therefore PDUs directed at those layers will be as simple as possible (minimum repercussions).

Fig 4 shows the test procedure for testing layer n in an IUT consisting of layers n-1 to n+1. In order to test layer n (layer n-1 having already been tested) a simple PDU of layer n+1 is embedded as user-data of the PDU under test for layer n. This n-layer PDU is then embedded as user-data in a (previously tested) n-1 layer PDU.

Essentially the method consists of simple padding and stripping of PDUs around the test data sent and received.

The ISO/IEC 9646 methodology for embedded testing provides the correct basis for looking at testing of embedded security software though there may be special considerations in this case. E.g. where a security function is used beneath the layer under test, should evaluation of the mechanism be carried out before conformance testing of the protocol is attempted? To what extent can black box testing of a mechanism be carried out, is that a legitimate part of conformance testing and is such testing sufficient to allow the conformance testing of the protocol to be carried out? These questions suggest that conformance testing and evaluation of a security implementation are inextricably linked. Conformance testing may be nested within the evaluation procedure. If a security mechanism hashes or encrypts data then PDU's recieved may not be checkable.

If the whole of an implementation is available for scrutiny and evaluation (ie there are no hidden parts) then the usual techniques for security testing are relevant. Embedded testing is not

applicable to such a system since direct access is available to all code for both static and dynamic testing.

If the implementation is divided into a TCB and non-security related functionality so that only part of the implementation is available for scrutiny then the situation is slightly different. For static analysis, embedded testing techniques have no relevance since all security related code is available for scrutiny.

Dynamic testing of such a system is more complex. Security related software must be directly accessible for evaluation purposes therefore an interface between the TCB and the non-trusted software must be provided. This is where the complexity lies - in defining this interface. If that interface can be defined and the security related software can thus be separated out then it is no longer embedded and therefore usual techniques for testing apply with particular attention to flow of information across the interface. If the separation cannot be made then the system as a whole has to be made available. In either case embedded testing has no relevance. Penetration testing is a special case of dynamic testing where embedded techniques have to be used. This is true white box testing - where knowledge of the design of the implementation directs testing at potential vulnerabilities. The security functionality is not separated out but is tested in situ through the non-trusted component(s). There is no standardised methodology for penetration testing as yet so this may be an area for further study using knowledge of the embedded testing techniques defined in 9646 as a guide.

Embedded testing is generally of relevance to conformance testing, in particular for protocol testing though mechanism testing may also be possible. It has more limited applicability to evaluation, solely in the area of penetration testing.

6.3.2 Testing modules in an embedded system

It has been assumed so far that the embedded system is being tested as a complete, single module, but it is in fact likely to be made up of smaller modules as shown in Fig 5, which each have a defined contribution to the whole system.

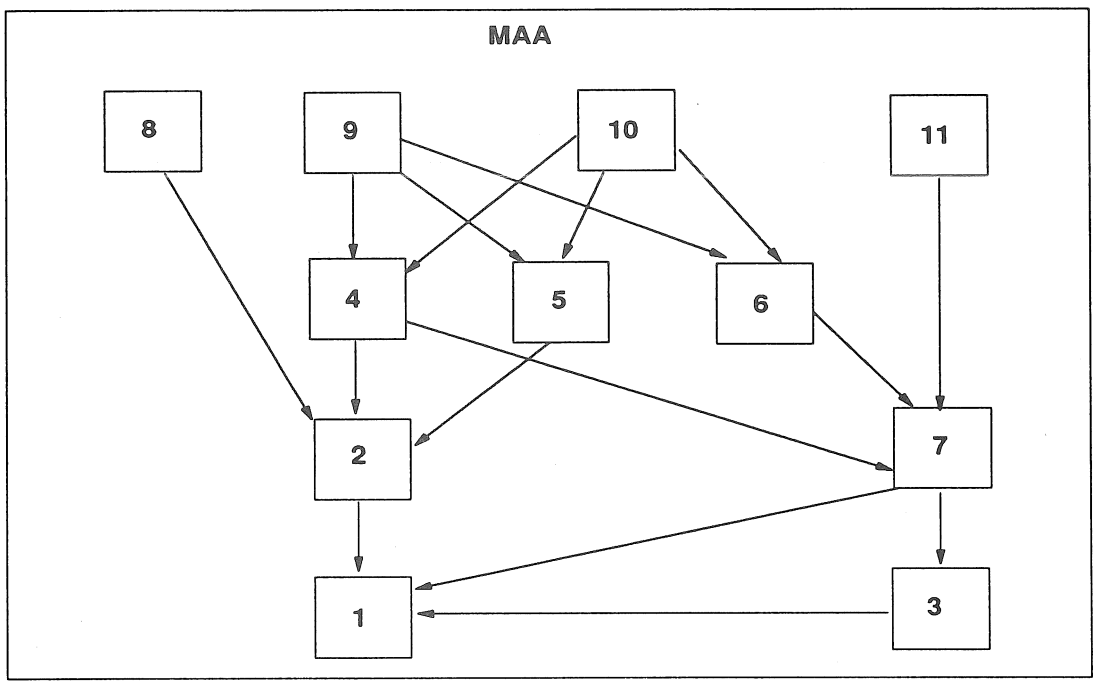


Fig 5: A representation of an entity as a number of modules.

It is generally easier to prove that each of these small modules is correct when using for example, formal methods, than to prove that the whole of the implementation is correct by carrying out the process on the entire implementation in one go. The application of unit testing, where individual modules are tested, and then integration testing, where each of the (previously tested) modules are tested together in the final configuration is a well established testing procedure: when it has been completed, the implementation is deemed to be correct. There are software tools available which are able to produce diagrams of the type shown in Fig 5 directly from the code: these are useful in giving a visual identification of the structure of the implementation, and for checking whether the functions shown to be in the embedded system correspond with those stated in the specification.

6.3.3 Trusted paths

Untrusted software such as a word processor may have to interact with the trusted component and this leads to the concept of a trusted path between the trusted and untrusted components. This concept is summarised in [10] as follows:-

> SMITE's trusted path is a user interface that is guaranteed to do what it is told and only what it is told. In other words it is Trojan horse free. It is therefore guaranteed not to exploit any signalling channels and to act faithfully on behalf of the human user ... For SMITE it is the normal means of communication with the system and has been designed as a useful user interface/editor. However all excursions off the trusted path should be monitored and the highest classification of information that the untrusted software has had access to will be maintained. This is generally known as a "high water mark" [8].

Where access to untrusted software occurs, the interface should be investigated during testing to ascertain if any security maintaining procedures are present of the type described in the above extract. If they are absent, then security can not be guaranteed (unless the bandwidth of covert channels is reduced to acceptable level and the integrity of data is not affected) whenever this interface is used.

A paper [32], which is concerned with covert, or hidden channels in such communication paths between untrusted and trusted software, describes a system using information flow analysis to discover potential hidden channels in the secure kernel of an operating system. It is said that this method may be generalised for wider use. Another paper concerning the detection of hidden or covert channels is [16], which states that the way in which such channels are utilised depends on the programming language and operating system used. Unfortunately, the usual testing techniques for detecting covert code may not always reveal the presence of covert channels and may require a specific type of analysis of the whole system.

Conclusions

It is unlikely that, at present, we can completely guarantee that all security requirements are met in an implementation, though the methods suggested in this report go some way towards achieving this aim. We therefore have to set realistic objectives as to what is practically feasible with current levels of technology and understanding. We have pointed out in this report areas which are particularly relevant to security evaluation and Strict Conformance Testing, suggesting areas of research which may be of value in resolving these issues.

What is required is a standardised specification system and test suite. Formal methods go some way towards the former, though as we have noted in section 3, security standards may require additional data in the form of conformance or security clauses. The latter, standardised test suites should follow from a more rigorous notation and the use of such clauses. This report has suggested a number of test methods that may be of value.

Once standardisation is in place this should lead to a greater understanding of the requirements of security evaluation leading to a test methodology that is both repeatable and reproducible. This should in turn allow comparability of results and a mutual recognition of test reports.

Acknowledgements

This report was written as part of the programme of work by the Data Security Group at NPL which is sponsored by the Information and Manufacturing Technology Division of the Department of Trade and Industry.

References

[1] Bennett et al. *Reverse Engineering Handbook.* ESPRIT II REDO project document 2487-TN-WL-1027. Version 0.3 (September 1990).

[2] Booch G. *Software Engineering with Ada.* Second edition (1986). The Benjamin/Cummings Publishing Company, Inc.

[3] Bowen Jonathan. *From Programs to Object Code and Back again using Logic Programming.* Draft Technical Note, Based on ESPRIT REDO document id. 2487-TN-PRG-1044. Issued for ProCoS December 1990 workshop.

[4] Chikosky E J and Cross II J H. *Reverse engineering and design recovery - a taxonomy.* IEEE Software 7(1) pp 13-17, January 1990.

[5] Cornwell Mark R. *A Software Engineering Approach to Designing Trustworthy Software.* Proc. of 1989 IEEE Symposium on Security and Privacy, pp 148-156, IEE Computer Society Press.

[6] Cranfield IT Institute. *Formal Description Techniques and Security Standard Conformance Testing. A report from the Cranfield IT Institute Ltd and commissioned by the Data Security Group.* NPL Report DITC 175/91, March 1991.

[7] Cullyer W J, Goodenough S J and Wichmann B A. *The choice of computer languages for use in safety-critical systems.* Software Engineering Journal, March 1991.

[8] Cummings P T et al. *Compartmental mode workstation: results through prototyping.* Proc IEEE Symp. on Security and Privacy, IEE Computer Society Press, April 1987.

[9] De Millo R.A, Lipton R.J & Perlis A.J. *Social Processes and Proofs of Theorems and Programs.* CACM Vol 22 No 5, May 1979

[10] Harrold C L. *An Introduction to the SMITE Approach to Secure Computing.* Computers and Security, Vol 8, No 8, October 1989.

[11] Harry A. *The use of formal methods in data security standards.* NPL Report DITC 205/92. August 1992.

[12] Harry A. *VDM specification of the MD4 message digest algorithm.* NPL Report DITC 204/92. August 1992.

[13] Harry A. *RAISE specification of the MD4 message digest algorithm.* NPL Report DITC 206/92. September 1992.

[14] Jacob J. *Basic theorems about security. Journal of Computer Security.* To be published.

[15] Jacob J & O' Halloran. *A notation for expressing confidentiality specifications.* Working draft.

[16] Jacob J. *Separability and the Detection of Hidden Channels*. Information Processing Letters 34 (1990) 27-29.

[17] Katz S & Manna Z. *Logical Analysis of Programs*. CACM 19,4, April 1976.

[18] Lai M K F. *Using the Modula-2 Standard to verify Modula-2 programs*. NPL Report DITC 183/91, June 1991.

[19] Lai M K F. *A Formal Interpretation of the MAA Standard in Z*. NPL Report DITC 184/91, June 1991.

[20] Lampard Richard. *Formal methods in IT security standards - scope for usage*. NPL Report DITC 177/91, March 1991.

[21] Lampard Richard. *Formal Description Techniques in Data Security: An Evaluation and Comparison*. NPL Report DITC 182/91, April 1991.

[22] Lampard Richard. *Specification of the Two-way handshake in VDM*. NPL Report DITC 188/91, September 1991.

[23] Lampard Richard. *An implementation of MAA from a VDM specification*. NPL Technical Memorandum DITC 50/91.

[24] Lampard Richard *Towards a Security Conformance Testing Framework for IT*. NPL Report (To be published).

[25] Lano K C and Breur P T. *From Code to Z Specifications*. ESPRIT II REDO project document 2487-TN-PRG-1029. Version 0.3 (1990).

[26] Manning Roger. *The Representation of Data Security Standards in SDL*. NPL Report DITC 190/91, September 1991.

[27] McDermott John. *A Technique for Removing an Important Class of Trojan Horse from High Order Languages*. Proc. of 11th National Computer Security Conference (NBS/NCSC) pp 114-117, October 1988.

[28] Munster Harold B. *LOTOS Specification of the MAA Standard, with an evaluation of LOTOS*. NPL Report DITC 191/91, September 1991.

[29] Parkin G I and O'Neill G. *Specification of the MAA Standard in VDM*. NPL Report DITC 160/90, February 1990.

[30] Szczgiel B.M & Harry A.C. *Security clauses in the writing and analysis of standards*. NPL Report 1992

[31] Thompson Ken. *Reflections on Trusting Trust*. CACM, Vol 27, No 8 August 1984.

[32] Tsai C R, Gligor V D and Chandersekaran C S. *On the identification of covert storage channels in secure systems*. IEEE Trans on Software Engineering, Vol 16 No 6 June 1990.

[33] UK Ministry of Defence. *The procurement of safety-critical software in defence equipment.* Interim defence standard 00-55, Issue 1, April 1991.

[34] Wichmann B A. *Insecurities in the Ada programming language.* NPL Report, DITC 137/89, January 1989.

[35] Williams F & Green S. *Software stamping*, NPL Report DITC 198/92.

[36] Wood A W. *Using Ada for embedded secure systems.* Proc. IFIP. TC-11 SEC '91. Seventh international conference and exhibition on Information Security, May 15-17 1991.

[37] Zave P. *An operational approach to requirements specification for embedded systems.* IEEE Transactions on software engineering. Vol SE-8. No 3. May 1982.

[38] *Ada user's group chairman dismisses security risk flaws.* Government Computing, Page 4, July-August 1991.

[39] ISO/IEC 9646. *Information technology - Open Systems Interconnection - Conformance testing methodology and framework.* October 1990.

[40] ISO/IEC 8731-2 (E). *Banking - Approved algorithm for message authentication - Part 2: Message authentication algorithms.* 1991.

[41] ITSEC. *Information Technology Security Evaluation Criteria.* Version 1.2 June 1991.

[42] ITSEM. *Information Security Evaluation Methodology.* Draft version 1.1 1992.

Glossary

Black box testing:

Is used for testing both hardware and software and establishes whether a product satisfies the specified external requirements and constraints without exploiting knowledge of its internal workings.

Dynamic analysis:

Analysis of software whilst it is executing. May need considerable instrumentation code which may be added by suitable software tools or manually.

Penetration testing:

Penetration testing is carried out when all the other types of testing have been completed and is designed to identify security weaknesses in the complete system which may not have been revealed by other tests. Whereas earlier tests were carried out to verify that the system performed to its specification and contained no covert code, penetration testing examines how the system behaves in conjunction with its associated hardware and system software and attempts to defeat the security controls of the system.

It is defined in the ITSEC glossary is:-

...tests performed by an evaluator on the Target of Evaluation (TOE) in order to confirm whether or not known vulnerabilities are exploited in practice.

and in the TCSEC (Orange Book)[4] as:-

The portion of security testing in which the penetrators attempt to circumvent the security features of a system. The penetrators may be assumed to use all system design and implementation documentation, which may include listings of system source code, manuals and circuit diagrams. The penetrators work under no constraints other than those that would be applied to ordinary users.

Such penetrators are often described as being members of a "tiger team" which is given the task of trying to compromise the system.

Semantics

The meaning attached to words or symbols in language statements.

Static analysis:

Analysis of the software code by examination rather than execution.

---

[4] TCSEC, Department of Defense Trusted Computer System Evaluation Criteria. US Department of Defense Standard (Orange Book), December 1985.

Syntax

The set of rules for combining the elements of a language (e.g. characters) into permitted constructions (e.g. keywords or program statements). The set of rules does not define the meaning.

Target of Evaluation (TOE):

(From the ITSEC glossary):-

An IT system or product which is subjected to security evaluation.

Trusted computing base (TCB):

As defined in the ITSEC glossary [41] is:-

The totality of protection mechanisms within a computer system, including hardware, firmware and software, the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that together enforce a unified security policy over a product or system. The ability of a TCB to enforce correctly a unified security policy depends solely on the mechanisms within the TCB and on the correct input by system administration personnel of parameters (e.g. a user's clearance level) related to the security policy.

And for the TCSEC is:-

The heart of a trusted computer system is the TCB which contains all of the elements of the system responsible supporting the security policy and supporting the isolation of objects (code and data) on which the protection is based. The bounds of the TCB equate to the security perimeter referenced in some computer security literature. In the interest of understandable and maintainable protection, a TCB should be as simple as possible consistent with the function it has to perform. Thus, the TCB includes hardware, firmware and software critical to protection and must be designed and implemented such that system elements excluded from it need not be trusted to maintain protection. Identification of the interface and elements of the TCB along with their correct functionality therefore forms the basis for evaluation.

Trusted software:

The software in a system which is security relevant.

Untrusted software:

The software in a system which is not security relevant.

White box testing:

Establishes whether a software product satisfies the specified requirements and constraints by exploiting knowledge of its internal workings.

Appendix 1: Further reading

This section contains books and papers containing further information on topics contained in this report.

Ames Stanley R Jr, Gasser Morrie. *Security Kernel Design and Implementation: An Introduction.* IEEE Computer 1983.

Cooling J E. *Software Design for Real-time Systems*. Published by Chapman and Hall. ISBN 0-412-34180-8. (This book covers many of the topics covered in this report. A comprehensive review of this book is given in *The Journal of Software Testing, Verification and Reliability,* Vol 1, No 2, July-September 1991).

Graham Dorothy. *Computer Aided Software Testing CAST Report*. Unicom Seminars Ltd.

Landwehr Carl E. *The Best Available Technologies for Computer Security*. IEEE Computer, July 1983.

Lapid Yeheskel, Ahituv Niv and Neumann Seev. *Approaches to Handling "Trojan Horse" Threats*. Computers and Security 5, 1986.

Sowerbuts Barry. *Formal Methods - White Hope or Red Herring?* SafetyNet, Apr/May/June 1989.

Sennet Chris. *High Integrity Software*. Pitman Publishing ISBN 0 273 03000 0 (paperback).