Enforcing Security and Safety Models with an Information Flow Analysis Tool

Roderick Chapman
Praxis Critical Systems Ltd.
20 Manvers Street
Bath BA1 1PX, United Kingdom
rod.chapman@praxis-cs.co.uk

Adrian Hilton
Praxis Critical Systems Ltd.
20 Manvers Street
Bath BA1 1PX, United Kingdom
adrian.hilton@praxis-cs.co.uk

ABSTRACT

Existing security models require that information of a given security level be prevented from "leaking" into lower-security information. High-security applications must be demonstrably free of such leaks, but such demonstration may require substantial manual analysis. Other authors have argued that the natural way to enforce these models automatically is with information-flow analysis, but have not shown this to be practicable for general purpose programming languages in current use.

Modern safety-critical systems can contain software components with differing safety integrity levels, potentially operating in the same address space. This case poses problems similar to systems with differing security levels; failure to show separation of data may require the entire system to be validated at the higher integrity level.

In this paper we show how the information flow model enforced by the SPARK Examiner provides support for enforcing these security and safety models. We describe an extension to the SPARK variable annotations which allows the specification of a security or safety level for each state variable, and an extension to the SPARK analysis which automatically enforces a given information flow policy on a SPARK program.

Categories and Subject Descriptors

D2.4 [**Software Engineering**]: Software/Program verification; F3.1 [**Logics and Meanings of Programs**]: Specifying and verifying and reasoning about programs

General Terms

Design,Measurement,Security,Verification

Keywords

Information flow, SPARK Ada, Dolev-Yao, Bell-LaPadula, security, safety

SIGAda'04, November 14–18, 2004, Atlanta, Georgia, USA.

1. INTRODUCTION

Software is often used to develop security-critical applications. Some of these applications are required to manage information of different classification levels, ensuring that each user may only access the data for which he or she has adequate authorisation. This requirement has two components; the developer must produce a design and implementation which supports this model and its constraints, and the implementation must support *verification* that the constraints are never violated. It is not enough for the implementation to be correct; for it to be secure, it must be *seen* to be correct.

In this paper we examine the problem of developing and verifying a security-critical application containing this security structure (often termed *multi-level security*). We will analyse recent work on information flow and security and show how the information flow analysis of the SPARK Examiner tool is appropriate for solving this problem. We will show how it is also important for the efficient analysis of safety-critical systems. We will then describe modifications to the current definition of the SPARK Ada language[2] and the Examiner which permit complete validation of Ada programs against defined security models such as the Bell-LaPadula model[3].

We intend that the additional flow analysis features described here will appear in a future commercial Examiner release. Although we anticipate possible minor changes to the syntax and analysis performed, we expect the released version to implement the analysis described here in all substantial aspects.

2. EXISTING WORK

In this section we identify typical standards which will inform the development and verification of a security-critical or safety-critical application. We then analyse a survey paper on information flow and security, and compare its conclusions against the information flow model of the SPARK annotated subset of Ada95.

2.1 Standards

The Common Criteria for IT Security [5] specify the development and verification activities that are suitable for applications at differing levels of security. These Evaluation Assurance Levels (EALs) range from EAL-1 (lowest) to EAL-7 (highest). As the EAL number rises, the required activities become more rigorous; at the higher levels, formal specification and analysis of the software becomes required.

For safety-related applications there are similar concepts for the safety criticality of software; RTCA DO-178B[10] for civil avionics software defines criticality levels E (lowest) through to A (most critical). As one would expect, the required development and verification activities become increasingly more onerous (and expensive) with increasing criticality level. As a result, if an avionics system were to contain a "core" of critical functionality at Level A and a larger body of utility code at Level D then either the entire system would have to be developed and verified at Level A or a rigorous argument would have to be applied that the Level D code could not in any way affect the integrity of the Level A code.

Another notation for safety criticality is the Safety Integrity Level (SIL), described in IEC 61508[8]. SIL-1 is the lowest level of safety integrity, and SIL-4 the highest; DO-178B level A approximates to SIL-3/SIL-4 when comparing the development and verification activities required.

In this paper we assume that we are attempting to validate systems at EAL-5 or greater (for security) and RTCA Level B or greater (for safety). We are therefore required to provide a rigorous and comprehensive justification for any statements which we make about the separation of data. Therefore we now look at how such statements may be expressed.

2.2 Information Flow

"Information flow" as it applies to conventional imperative computer programs has a range of definitions, and is often confused with "data flow"; we shall take the definition as expressed in Barnes[2] p.13:

- data flow analysis is concerned with the *direction* of data flow; whereas
- information flow analysis also considers the *coupling* between variables.

An example of the difference between data flow and information flow comes from the following (Ada) code:

```
while (X < 4) loop
   A := A + 2 * B;
   X := X * 2;
end loop;
```

Here, data flow analysis would state that data flows from X to X and from A and B to A. Information flow analysis would note additionally that the final value of A is affected by the initial value of X, and hence that there is information flow from X to A.

Sabelfeld and Myers[11] recently surveyed the use of information flow analysis in software. They viewed the fundamental problem of maintaining security as one of tracking the flow of information in computing systems. They examined the various ways that secure information could leak into less secure information, overtly and covertly. They identify in particular the concept of *implicit* information flows, an example of which is given in the `while`-loop example above.

Their survey characterised language-based information flow as requiring:

1. semantics-based security, so that rigorous argument could be made about a variation of a high-security value not affecting a low-security value; and
2. a security type system, so that a program or expression can be assigned security values (types) and the type changes of the program or expression can be characterised.

They concluded that "standard" security practices do not and cannot enforce the end-to-end confidentiality required by common security models. They characterised the existing work in security and information flow analysis, but notably did not address the work of Bergeretti and Carré[4].

2.3 Security models

The Bell-LaPadula (BLP) model of computer security[3] enforces two properties:

1. no process may read data from a higher security level; and
2. no process may write data to a lower security level.

Such multi-level security has a number of problems; Anderson[1] provides a list of them including:

- "blind write-up": the inability to inform low-security data whether a write to high-security data has happened correctly;
- "downgrading": moving information from a high security level to a lower level is sometimes desirable; and
- "TCB bloat": a large subset of the operating system may end up in the Trusted Computing Base (TCB).

The Dolev-Yao security model[6] makes secret information indivisible; it cannot be leaked in part but only in total.

2.4 Information Flow in Ada

Bergeretti and Carré wrote a seminal paper[4] describing a practical implementation of information flow analysis in the SPADE Pascal language (although the principles were applicable to most conventional imperative programming languages). This was managed by the composition of matrices representing information flow dependencies between variable imports and exports. Notably, conditional and infinite loops were permissible and analysable within the framework of such a language; these were managed by computing the transitive closure of the information flow matrix corresponding to one execution of the loop.

This information flow model was implemented in the SPARK annotated Ada subset[2]. The subset requires each subprogram to be annotated with the required information flow ("derives" annotations) if information flow analysis is required. The SPARK subset is enforced by the SPARK Examiner tool which checks the required information flow of each subprogram against the actual information flow. The annotations (and other SPARK rules such as the ban on circular dependency) are necessary to make this information flow analysis tractable.

The result of this is that it is possible, in a fully-analysed SPARK program, to be certain that a given exported variable is independent of a given imported variable. This is a key step towards supporting multi-level security in SPARK Ada, and makes it easier to write demonstrably secure and correct code, but is not a complete solution. In Section 3 we will describe how SPARK Examiner analysis may be extended better to implement these checks.

We now examine case studies of the use of SPARK, and the utility of information flow in real applications.

2.5 Security and Safety Applications

An example of a high-security application which was partly developed in SPARK is the MULTOS CA[7]. This was developed and verified at a high level of integrity, approximating to the Common Criteria EAL-5. The delivered system had a defect rate of 0.04 errors per thousand lines of code (KLOC). This showed that SPARK was a practical language for implementing high-security applications. Part of the analysis required during development and verification of the CA was to show that secret data could not leak out directly or indirectly in unclassified outputs.

An example of a safety-critical application with mixed integrity levels where information flow analysis was helpful was the SHOLIS information system described by King et al.[9]. This application mixed SIL-4 and non-critical code, and was written in the SPARK subset of Ada83. Static verification was used, including information flow analysis and partial program proof, to verify significant properties of SHOLIS. There was a successful argument based partly on information flow analysis that the high-SIL code was not compromised by the low-SIL code; however, this argument had to be made manually, based on the validated information flow of each subprogram, as the Examiner did not provide such tracing facilities at the program level.

3. IMPROVING SPARK ANALYSIS

The outstanding need in SPARK is to be able to mark state variables in packages with a (numerical) level of security and / or safety. This has been implemented by allowing an optional aggregate after a package ("own") variable declaration.

3.1 Marking security levels

Suppose that a package `Classify` was defined as follows to create various secrecy levels:

```
package Classify is
   -- Security levels
   UNCLASSIFIED : constant := 0;
   RESTRICTED   : constant := 1;
   CONFIDENTIAL : constant := 2;
   SECRET       : constant := 3;
   TOPSECRET    : constant := 4;
end Classify;
```

Then we define a package `KeyStore` to store and manage a symmetric encryption key `SymmetricKey`, designed to mutate after each encryption according to a rotation parameter `RotorValue`. The key is clearly a high-security data item, with the rotor value requiring less security.

`KeyStore` marks its state variables with the field `Integrity`[1] thus:

```
--# inherit Classify;
package KeyStore
--# own SymmetricKey(Integrity => Classify.SECRET);
--#   RotorValue(Integrity => Classify.RESTRICTED);
is
   procedure Rotate;
   --# global in RotorValue;
   --#    in out SymmetricKey;
   --# derives SymmetricKey from
   --#    SymmetricKey, RotorValue;

   procedure Encrypt(C : in MessageBlock;
                     E : out MessageBlock);
   --# global in SymmetricKey;
   --# derives E from C, SymmetricKey;
   ...
end KeyStore;
```

[1] `Integrity` was chosen to make sense for both security and safety applications

Any security analysis must show, in the case of this code, that `SymmetricKey` data cannot leak into `RotorValue` or data; i.e. there must be no subprogram (or main program) whose information annotation shows `SymmetricKey` as an import to `RotorValue`.

Note that in this case package `Classify` will need to be inherited by all relevant package, but will never be withed and so never compiled.

3.2 Implementation

We now define how the integrity levels are marked in the SPARK language, and how they are enforced by the SPARK Examiner.

The extra information flow checking is invoked using the `/policy=X` command line switch to the Examiner. Current valid policy values are `security` and `safety`.

3.2.1 *Variable declaration*

The above `Integrity` property on package own variables is a static property; at any point in static analysis of the code, the actual and required integrity level of an own variable is known. Own variables without `Integrity` levels are taken to have a default integrity level of `Natural'Last` (i.e. very highly classified) if imported under a security policy, and `Natural'First` (i.e. unclassified) if exported under a security policy. This gives the most paranoid checking so that data may not leak from an input to an output through an intermediate variable with unspecified integrity.

If the analysis policy is `safety` then the default integrity values for unspecified own variables are reversed.

3.2.2 *Integrity checks*

Information flow is declared explicitly in `derives` annotations for subprograms; an example from our cryptographic `KeyStore` is the `Rotate` subprogram which changes the key based on the value of `RotorValue`:

```
procedure Rotate;
--# global in RotorValue;
--#   in out SymmetricKey;
--# derives SymmetricKey from
--#   SymmetricKey, RotorValue;
```

Using policy `security` the Examiner will then check that the integrity level of the export `SymmetricKey` (`SECRET`) is *no less* than the integrity levels of any import (`SECRET` and `RESTRICTED`). With policy `safety` the checks would be that the integrity of the export is *no more* than the integrity levels of any import, i.e. that high-safety data exports cannot be contaminated by low-safety data imports.

Information flow is also checked at each procedure call by substituting in actual parameters (which may be own variables) for formal parameters and rechecking the procedure's known information flow for integrity flows. As examples we

give the procedures which get and set the rotor and key values:

```
procedure GetKey(This_Key : out Key);
--# global in SymmetricKey;
--# derives This_Key from SymmetricKey;

procedure GetRotor(This_Rotor : out Rotor);
--# global in RotorValue;
--# derives This_Rotor from RotorValue;

procedure SetKey(New_Key : in Key);
--# global out SymmetricKey;
--# derives SymmetricKey from New_Key;

procedure SetRotor(New_Rotor : in Rotor);
--# global out RotorValue;
--# derives RotorValue from New_Rotor;
```

Because `RotorValue` is `RESTRICTED` and `SymmetricKey` is `SECRET`, the analysis requires a check at each invocation of these subprograms that the actual parameters mapped to formal parameters do not violate integrity flow checks.

3.2.3 Analysis techniques not adopted

One analysis technique which we considered (but rejected) was to track the integrity flows within each subprogram body and raise warnings at each individual statement where a violation occurs. We now explain how this would have worked and why we rejected it.

In the following code, the programmer generates a key and rotor, encrypts a message with it, then tries to create a new rotor based on the old key.

```
-- Key and rotor generation
R1 := KeyStore.MakeRotor(34,56,22,55);
KeyStore.SetRotor(R1);
K1 := KeyStore.MakeKey(66,11,2,4);
KeyStore.SetKey(K1);
-- Encrypt a message
KeyStore.Encrypt(C => Clear, E => Encrypted);
-- Get a copy of the (changed) key and break
--  it down into data
KeyStore.GetKey(K1);
KeyStore.DecomposeKey(K1,I1,I2,I3,I4);
-- Build a new rotor
R1 := KeyStore.MakeRotor(I1,I2,I3,I4);
KeyStore.SetRotor(R1);
```

The statement-by-statement information flow would proceed as shown in Table 1, where `C` denotes `Clear`, `RV` denotes `RotorValue`, `SK` denotes `SymmetricKey`, and `E` denotes `Encrypted`.

The final step causes `RotorValue` to exceed its assigned integrity level, and would generate a static integrity flow error.

The problem with this technique arises from the need to track the integrity levels of local variables. It quickly becomes clear that for practical analysis reasons each local variable involved needs to be assigned an integrity level. This is possible, and is done by declaring them as own variables with integrity levels in a package embedded in the subprogram in question, but is cumbersome. It also requires substantial rework of any existing code which we may want to retro-analyse.

R1	K1	SK	RV	I1	C	E	Instruction
0	0	-	-	-	1	-	–
0	0	-	-	-	1	-	R1 :=
0	0	-	**1**	-	1	-	SetRotor
0	**0**	-	1	-	1	-	K1 :=
0	0	**3**	1	-	1	-	SetKey
0	0	3	1	-	1	**3**	Encrypt
0	**3**	3	1	-	1	3	GetKey
0	3	3	1	**3**	1	3	Decompose
3	3	3	1	3	1	3	R1 :=
3	3	3	**3**	3	1	3	SetRotor

Table 1: Information flow for example

3.2.4 Example of checking

We placed the code described above into procedure `Operate` in a package `Crypto` and annotated the declaration with the correct `derives` annotation thus:

```
--# inherit KeyStore,Classify,BitString;
package Crypto
  --# own Clear,
  --#  Encrypted(Integrity => Classify.SECRET);
is
   procedure Operate;
   --# global out KeyStore.RotorValue,Encrypted;
   --#        in out KeyStore.SymmetricKey;
   --#        in Clear;
   --# derives
   --#   KeyStore.SymmetricKey,
   --#   KeyStore.RotorValue
   --#   from
   --#     KeyStore.SymmetricKey
   --#   &
   --#   Encrypted
   --#   from
   --#     Clear,
   --#     KeyStore.SymmetricKey
   --#   ;
end Crypto;
```

Analysis using `/policy=security` gives the following static semantic error:

```
Unit name:  Crypto
Unit type:  package specification
Unit has been analysed, any errors are listed below

1 error(s) or warning(s)

Line
  30     --#     KeyStore.SymmetricKey
                 ^1
*** (  1) Semantic Error    :175: Information flow
          from KeyStore.SymmetricKey to
          KeyStore.RotorValue violates the
          selected information flow policy..
```

which correctly identifies the potential leak.

3.3 Case study

SHOLIS is the Royal Navy's Ship Helicopter Operating Limits Information System [9] designed to assist landing of

helicopters on Royal Navy Type 23 frigates. Failure of this system could result in the death of helicopter pilots and passengers, loss of a helicopter and damage to the ship. This is intolerable for normal operation, hence SIL-4 reliability is required to give sufficient confidence that such an accident will not happen during the in-service lifetime of the system. SHOLIS is an on-demand system rather than a continuously operating system, and so has a required probability of failure to function on demand of approximately 10^{-4}; a more precise probability would be specified in the system safety case.

However, the bulk of the SHOLIS code does not relate to critical system functionality. The code specific to SIL-4 must be analysed at a deep level; the rest of the code can be analysed less deeply *as long as* it can be shown not to affect the SIL-4 data adversely.

3.3.1 Original analysis

The original SHOLIS code is Ada 83 and consists of 75 source files and shadow files amenable to SPARK analysis; 26.5KLoC of non-blank non-comment non-annotation code. It is a substantial program and therefore a suitable test to see if integrity level checking scales.

Without any own variables annotated, a full SPARK analysis by an Examiner with `policy=safety` generated no integrity errors, as we would expect.

3.3.2 Identifying critical outputs

To enable easy marking of variable safety criticality we added a single package `Safety`:

```
package Safety is
  NSC : constant := 0;
  SC  : constant := 1;
end Safety;
```

We took as an example the safety-critical output `Alarm2Z` in a package `RMR` which represents an alarm signal on the front panel. This was annotated:

```
--# own Alarm2Z : BasicTypes.OkFailT
--#     (Integrity => Safety.SC);
```

A re-analysis of package `RMR` raised no integrity errors. The other package that used `Alarm2Z` was `EVENT` which was the main event handler. A SPARK of this package raised integrity flow errors in subprogram `Sync`, where `Alarm2Z` depended on a large range of other inputs, which had not yet been marked as safety-critical. This was what we would expect so far and confirmed that basic safety integrity analysis was working.

3.3.3 Extending analysis

The next phase of work started in the `SENSOR` package which was near the middle of the package hierarchy. We set the package variables representing current speed, heading, roll, pitch and wind velocity state to be safety-critical and then ran a trial SPARK analysis. This indicated many own variables in this and other packages which caused integrity flow errors since they had no explicit integrity level.

For each of these packages in turn, we:

1. marked all of the package own variables as non-critical (`Safety.NSC`);
2. re-analysed the package specification;
3. analysed the package body to ensure that there were no integrity flow errors at subprogram call points; and
4. if necessary, transformed `NSC` variables to `SC` status and re-ran.

Eventually we converged on a stable SC/NSC partition of the variables.

3.3.4 Declassification

The `DisplayBuffers` state variable in I/O package `sio5` was a point where safety-critical and non-safety critical data merged. It was necessary during the actual project to produce a manual justification that the buffer would never be filled with non-safety critical data when there was safety-critical data to be added and displayed to a user. We therefore set its integrity to `NSC` and ignored all integrity errors relating to flows from `SIO5.DisplayBuffers`.

3.3.5 Results

There were 1282 integrity flow errors, but every single one of these referred to a flow from `SIO5.DisplayBuffers` to `Fault.Faults`, as expected. Therefore only one manual argument is needed to validate the separation of `SC` and `NSC` data at this level.

Of the 233 package specification variables which were given integrity levels, 110 were `NSC` and 123 were `SC`.

3.3.6 Lessons learned

SHOLIS was developed using a now out-of-date version of the SPARK Examiner which did not support proof work involving abstract state; as a consequence there were many more public own variables than you would expect in a well-designed modern SPARK program. This made the conversion work slower; at the top level, as noted above, it increased the time required beyond what was available for the study.

The "TCB bloat" problem noted in Section 2.3 did not seem to be a problem. While working up the calling hierarchy there was a small amount of returning to lower levels to make `NSC` variables into `SC`, but this did not spread out of control.

There is a clear need for a declassification mechanism, as discussed in more detail in Section 4.3. Being able to suppress the integrity flow errors from `DisplayBuffers` would have made the transformation process easier.

3.4 Possible tactical extensions

Given the preceding work, it is relatively simple to extend the own variable annotation to allow other fields in the aggregate. Within the security domain, there are considerations which mean that security cannot be considered on a linear scale.

An example is a set of documents on an international military aviation development where markings might include NATO RESTRICTED, UK RESTRICTED and ICELAND RESTRICTED. They are all at the same level of security, but apply to different nationalities in different ways. A UK citizen could receive the first two, a German could receive the first one only, and a Russian could not receive any. The nationality information could be represented by an additional field, which might be an array of booleans mapping each country code to an Allowed/Forbidden code.

3.5 Security policies

So far we have considered enforcing the Bell-LaPadula security policy. However, there are other policies which may be desirable for enforcement. One example is a policy where information at security level N may only flow into other information at security level N; there is no concept of ordering on these security levels, they may simply not be mixed.

There is further work to be done on investigating whether other information flow policies are desirable and useful for security-critical or safety-critical code. In principle they should not be complicated to enforce. The `/policy=X` command line switch provides a hook to specify different policies in future.

4. ISSUES FOR FUTURE RESEARCH

In this section we discuss the limitations of the existing work and examine how the analysis techniques may be extended in the future.

4.1 Difficulties with analysis

The concept of "label creep" as identified by Sabelfeld and Myers refers to the tendency of data to increase in security level as program execution progresses; assigning a SECRET value to one field of a large CONFIDENTIAL record will make the entire record SECRET . It remains to be seen how SPARK security programs should be designed in order to minimise label creep.

Concurrency is more complex because there is the possibility that security information may leak from the observable scheduling order of the processes. This is addressed to some extent because SPARK analysis of Ravenscar programs does information flow across tasks.

4.2 Wider analysis

One extension suggestion that has come from a software development project within the company is the idea of *subprogram* integrity level. The motivation is similar to that for the SHOLIS analysis; that only part of a given program is safety-critical, and that verification activities (proof, unit testing levels, coverage analysis etc.) may be better focused on the safety-critical parts. Subprograms are a more useful unit of division for these activities than package state.

The algorithm for identifying a subprogram's actual integrity level is to examine its exports. If it only exports own variables then the subprogram integrity level is the maximum of all exported own variable integrity levels. If some exported variables are formal parameters then each invocation of the subprogram must be examined for own variables that may be actual parameters, and the maximum integrity level taken over *all possible* exported own variables. Functions are taken to be equivalent to a procedure with a single exported parameter.

There are two clear choices for implementing this strategy:

1. whole-program analysis, determining subprogram integrity level once all invocations are known; or
2. partial-program analysis, annotating each critical subprogram declaration with its integrity level and checking at declaration and each invocation that the maximum integrity level is not violated.

The first choice is minimal-impact, but does not admit analysis until the whole program is written which is likely to prove troublesome; the integrity level of many subprograms will not be known until very late in the development process, by which time testing should already be ramped up.

The second choice is higher impact; it requires an extra annotation to be added to the SPARK language and checked by the Examiner, and requires developers to add the annotation to each potentially critical subprogram as it is specified. However, the benefits of the partial program analysis are likely to outweigh these drawbacks.

4.3 Subverting the analysis

Declassification is occasionally necessary in security programs; this is when the assigned security level of information is deliberately reduced. An example would be a security filter which took SECRET information, stripped out sensitive fields and output information which was no more than CONFIDENTIAL level. This can be done in SPARK by hiding the body of a declassifying subprogram from the Examiner, but this is clearly not an optimal solution. A better solution should be found.

4.4 Considerations for certification

In very high security applications it may be necessary to certify the object code as well as the source code. It remains to be seen whether and how the information known from the SPARK source code analysis can be carried over to inform an object code analysis.

There are other ways by which secure information can be observed, such as covert or timing channels. A full implementation of multi-level security information analysis should be followed by an analysis of how much information could be leaked this way.

5. CONCLUSIONS

In this paper we have described recent work on applying information flow analysis techniques to enforcing multi-level security in a single software application. We have shown how the requirements listed by Sabelfeld and Myers[11] are partially satisfied by the information flow analysis possible with SPARK Ada and the SPARK Examiner. We have further shown that the existing SPARK language and analysis may be extended to enforce the Bell-LaPadula security model with relatively little change.

SPARK Ada has already proven itself in high-security and safety-critical application development. It now appears to be an effective choice of language to partition data of differing criticality, and provide a low-cost but robust argument of safety or security for an application. SPARK already provides the semantics-based security required by Sabelfeld and Myers; the extensions to own variable annotations now provide the complementary security type system.

For end-to-end confidentiality in a secure system, we believe that SPARK Ada's extended information flow analysis provides a hard-to-refute justification of data security.

5.1 Acknowledgements

The authors are grateful to Peter Amey, Neil White and Will Ward from Praxis Critical Systems for their feedback on this paper and the prototype integrity checking facilities of the Examiner.

6. REFERENCES

[1] R. J. Anderson. *Security engineering: a guide to building dependable distributed systems*. Wiley Computer Publishing, 2001. ISBN 0-471-38922-6.

[2] J. Barnes. *High Integrity Software: The SPARK Approach to Safety And Security*. Addison Wesley, April 2003.

[3] D. E. Bell and L. LaPadula. Secure computer systems. Technical Report ESR-TR-73-278, Mitre Corporation, November 1973. v. I and II.

[4] J.-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.

[5] Common Criteria. *Common Criteria for Information Technology Security Evaluation*, August 1999.

[6] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, August 1983.

[7] A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, pages 18–25, Jan/Feb 2002.

[8] International Electrotechnical Commission. *IEC Standard 61508, Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems*, March 2000.

[9] S. King, J. Hammond, R. Chapman, and A. Pryor. Is proof more cost effective than testing? *IEEE Transactions on Software Engineering*, 26(8):675–686, August 2000.

[10] RTCA / EUROCAE. *RTCA DO-178B / EUROCAE ED-12: Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

[11] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.