Tokeneer: Beyond Formal Program Verification

Yannick Moy[1], Angela Wallenburg[2]
1: AdaCore, 46 rue d'Amsterdam, F-75009 Paris (France),
yannick.moy@adacore.com
2: Altran Praxis, 20 Manvers Street, Bath BA1 1PX (United Kingdom),
angela.wallenburg@altran-praxis.com

Abstract Tokeneer is a small-sized (10 kloc) security system which was formally developed and verified by Praxis at the request of NSA, using SPARK technology. Since its open-source release in 2008, only two problems were found, one by static analysis, one by code review. In this paper, we report on experiments where we systematically applied various static analysis tools (compiler, bug-finder, proof tools) and focused code reviews to all of the SPARK code (formally verified) and supporting Ada code (not formally verified) of the Tokeneer Project. We found 20 new problems overall, half of which are defects that could lead to a system failure should the system be used in its current state. Only two defects were found in SPARK code, which confirms the benefits of applying formal verification to reach higher levels of assurance. In order to leverage these benefits to code that is was not formally verified from the start, we propose to associate static analyses and dynamic analyses around a common expression of properties and constraints. This is the goal of starting project Hi-Lite, which involves AdaCore and Altran Praxis together with several industrial users and research labs.

Keywords System formal development, Verification and validation, Certification and dependability

1 Introduction

Formal verification consists of the application of mathematical techniques to verify properties of systems. Formal program verification targets systems at the level of source programs, where the semantics of the programming language chosen gives a precise meaning to the programs analyzed. Frameworks for formal program verification, such as the SPARK framework that we develop and promote at Altran Praxis and AdaCore, rest upon two crucial features: powerful software tools that automate most of the reasoning and fine-grain interactions with users through the manual specification of program properties.

Formal verification is desirable for critical systems and has been shown to be practically feasible. In the Tokeneer Project [1, 4] initiated by the NSA, Praxis developed a small-sized security system using formal verification methods and showed that it reached the higher assurance levels of the Common Criteria. Since the open-source release of the project [6], very few problems have been found in the system, despite scrutiny of research teams and individual security experts.

Still, two problems were found prior to the work we report here, and we found many more issues during this work. It may seem unintuitive that a formally verified system can contain problems. There are several reasons for why problems can be present in a formally verified system:

1. Formal verification is not applied to all parts of a system. Around a formally verified core, there are usually parts of the system that cannot be verified formally, because they deal with interfacing with other systems or a user, or because it would be too costly to verify them (effort/time/money). The cost/benefit ratio of formal verification is particularly important to take into account for those less critical parts.
2. Formal verification is only as strong as the specification is. When formally verifying the absence of runtime errors in programs, the specification is given by the semantics of the programming language. However, in general there is no way to "guess" the intended meaning of some part of a system, but the developer has to indicate it in a specification. Writing specifications can be as error-prone as writing programs. In practice, these specifications are not complete functional specifications, as this would be much too costly. Only the properties that a developer specified can be formally verified.
3. Formal verification techniques do not target all desirable properties of programs. Most formal verification frameworks verify the absence of run-time errors, and properties expressed by developers as invariants or function contracts. In particular, formal verification techniques are not usually good at detecting dead code and "strange" code (inefficient or unidiomatic code), or covert channels.
4. The formal verification method itself may be flawed, for example if the implementation of the underlying theorem prover or static analyzer contains problems.

In this paper, we examine in detail the problems that we found in Tokeneer since its delivery. We are interested here in why they were not found by formal verification (or testing, or code review), and how they can be found. We show that they can be found by a combination of code review directed at the “blind spots” of formal verification and improved tools or new tools that we present in this paper. This is the reason for our bold claim to go beyond formal program verification. Of course, formal program verification will always be a very sought-after goal. Here, we show how it can be complemented by other techniques.

We start in Section 2 with a presentation of the Tokeneer Project and we describe the associated formal verification effort. In Section 3, we describe the setup of our experiments to find problems in the SPARK and Ada code of the Tokeneer Project by static analysis and code review. We report the results of these experiments in Sections 4 and 5 as a list of defects and code quality issues that were found. We summarize and discuss these results in Section 6, before we describe in Section 7 how a new project, Hi-Lite, addresses the problem of associating formal verification with other techniques. Finally, we conclude in Section 8.

2 Tokeneer - A Successful Formal Verification Project

The Tokeneer Project was initiated by the NSA (National Security Agency) to demonstrate that developing highly secure systems to the level of rigor required by the higher assurance levels of the Common Criteria [3] was possible. The NSA asked Praxis to develop part of an existing secure system (the Tokeneer System, a biometrics identification station) in accordance with Praxis’ own Correctness by Construction development process [2].

This development and research work was completed in 2003 and made publicly available by the NSA and Praxis in October 2008, including the source code for the application. The overall development took 12 person-months. Source code is written in SPARK for the core (9939 lines of code + 6036 lines for data and flow annotations + 1999 lines for proof annotations) and Ada for the interfaces and drivers (3697 lines of code). SPARK is both a subset of Ada and a specification language that allows the expression of many program properties. The SPARK framework is based on powerful tools for automatically verifying most of these properties. Ada is a programming language heavily used for the development of critical applications, partly thanks to its intrinsic specification features, but there are no verification tools for Ada that compare to those which exist for SPARK.

The development process adopted by Praxis consists of a well-designed workflow from System Requirement Specification to Formal Specification to Formal Design to INFORMED Design to SPARK implementation. Here, we only consider the final result of this development process, which is the SPARK implementation, as well as the supporting Ada code. The assurance process adopted by Praxis consists of activities both at the specification and at the coding levels. The latter consists of testing, static analysis and formal proof activities. The static analysis was carried out automatically by SPARK Examiner which ensured that all of the SPARK code in Tokeneer respected data-flow properties and information-flow properties (as well as ensuring that all data is initialized before use). The formal proof activities were carried out automatically by SPARK Simplifier for a large part (93% of Verification Conditions, a.k.a. VCs) and by a user in the remaining cases (2% of VCs for which the user writes a proof script which is checked automatically by a SPARK tool called the Proof Checker and 5% of VCs manually reviewed) which ensured that all of the SPARK code in Tokeneer was free from run-time errors and that it respected the security properties specified. The two security properties verified were expressed as function contracts: 1) the property that the guard must have performed an “override lock” operation for the door to be unlocked; 2) the property that the alarm should be raised whenever the door is in an insecure state.

Here is an excerpt of a SPARK contract for procedure EnrolOp which progresses enrolment of a user with the biometric station:

```
procedure EnrolOp;
--# global in       State;
--#        in out KeyStore.State;
--# derives KeyStore.State from State,
--#                             KeyStore.State;
--# pre   EnrolmentIsInProgress(State) and not
--#         PrivateKeyPresent(KeyStore.State);
--# post  PrivateKeyPresent(KeyStore.State) <->
--#         not EnrolmentIsInProgress(State);
```

The data-flow part of this contract (introduced by the keyword **global**) expresses that the effects of calling EnrolOp depends on the values of global variables State and KeyStore.State, and that EnrolOp may change the value of KeyStore.State. The information-flow part of this contract (introduced by **derives**) expresses that the value of KeyStore.State when the procedure returns depends on the input values of both State and KeyStore.State. The precondition part of this contract (introduced by **pre**) expresses that EnrolOp should only be called in a context where enrolment is in progress and the key used is new. The postcondition part of this contract (introduced by **post**) expresses that when EnrolOp returns, either the key has been accepted and the enrolment is terminated, or the key has not been accepted and the enrolment is still in progress.

Besides formal verification, reviews were applied at all levels of the development process, including reviews of the SPARK code against Formal Design, INFORMED Design and Coding Standard. The system was also tested independently by a different group in another company, SPRE.

At the time the project was made public in 2008, zero problems had been found by the customer post delivery.

3 The Tokeneer Project under Scrutiny

Since the open-source release of the Tokeneer Project, its source code has been under the scrutiny of research teams and individual security experts, which resulted in the discovery of two problems prior to this work. We report here on these problems and on the several more that we have found by applying static analysis and code review on both the SPARK code and its supporting Ada code. We could not apply testing techniques (in particular coverage techniques) as the testing material (user interfaces and test-suite) was not available. It was developed by SPRE, not Praxis, and not released as open-source.

3.1 Experimental Setup for Static Analysis

We have applied the different tools developed at AdaCore and Praxis to the static analysis of either SPARK code or both Ada and SPARK code:

- The GNAT compiler for Ada (and SPARK), which issues simple warnings related to data flow;
- The static analyzer CodePeer, which performs a deep analysis of Ada (and SPARK) code to infer contracts and detect various kinds of errors (run-time errors, logic errors, concurrency errors);
- SPARK Examiner, SPARK Simplifier and SPARK Dead Path Analyzer (only for SPARK). SPARK Examiner performs in-depth data-flow and information-flow analysis. SPARK Simplifier proves by deduction that the code is free from certain classes of run-time errors and that the code meets its specification. SPARK Dead Path Analyzer is a new SPARK tool that finds paths that will never be run regardless of program inputs. Code that only has dead paths through it is dead code.

In particular, we applied here the latest versions of the GNAT compiler and SPARK tools which have extended verification capabilities compared to the versions used during the development of the Tokeneer Project in 2003. We used GNAT Pro 6.3.2 (with option -gnatwa), CodePeer 1.1 and SPARK Pro 9.0 (which includes SPARK Examiner, SPARK Simplifier and SPARK Dead Path Analyzer). The tools used during the Tokeneer Project development were GNAT 3.15p and SPARK 7.0.

Only a straightforward setup was needed to run any of the tools. In an attempt to get more precise ranking results with CodePeer (more true problems), we manually translated all SPARK preconditions and postconditions into the equivalent GNAT Precondition and Postcondition pragmas, which took half a day. These pragmas define contracts as Boolean expressions that should be verified at run-time. Both GNAT and CodePeer correctly treat these executable contracts. The results were quite unexpected: we did not get more precise ranking results, but CodePeer detected instead various redundant conjuncts in the Boolean formulas used for annotations. Since these experiments, we have disabled the generation of such redundant conjuncts in preconditions and postconditions, as they rarely point to real errors on usual programs. In the following, we refer to running CodePeer in special mode when the problem was found due to this feature. For each problem found by static analysis, we report the output of the tool that indicated a possible error, in order for the reader to judge for himself or herself the usability of these messages.

The run-time for these experiments was quite small: on an x86-64 box with two Intel Core 2 Duo CPUs (4 cores total) at 2.93GHz and 4 GB RAM running Ubuntu 9.10, compiling takes 2.6s, running SPARK Examiner takes 8.9s, running SPARK Simplifier takes 32s, and running SPARK Dead Path Analyzer takes 3mn 28s. Run-time for CodePeer depends on the mode in which it is used, which dictates mostly the number of partitions in which the code is split and separately analyzed. CodePeer takes 1mn 27s in quick mode (6 partitions), 2mn 10s in normal mode with partitions (3 partitions) and 17mn 15s in normal mode without partitions. Running CodePeer in quick mode was sufficient for all problems we report in the following but one, which is only detected by running CodePeer without partitions.

The time to review the results of each analysis was very different for each tool. It was immediate to check that SPARK Examiner did not issue error messages and that all VCs were either proved automatically by SPARK Simplifier or had a corresponding manual review (a SPARK tool called POGS for Proof ObliGation Summarizer performs this check automatically). It took a day to review the dead paths reported by SPARK Dead Path Analyzer (154 dead paths in 22 subprograms). It took a few minutes to review the compiler warnings (290 with option -gnatwa). It took a day to review all the high and medium warnings issued by CodePeer (84 in quick mode, 86 with partitions, 174 without partitions). We only looked at very few low warnings (80 in quick mode, 80 with partitions, 128 without partitions). The lower number of warnings when running CodePeer with partitions is due to the optimistic analysis of calls to functions across partitions.

Overall, it took three days to apply the static analysis tools and review the results.

3.2 Experimental Setup for Code Review

We decided to manually review three specific "blind spots" of the SPARK verification tools:

- Parts of SPARK code inside a hide annotation, which contain Ada code that does not fit in the SPARK subset;
- Possible non-termination of loops, as SPARK does not require that termination is proved;
- Handling of exceptions possibly raised by lower level code in Ada, as SPARK does not know about exceptions.

Code reviews were performed inside an IDE, GNAT Programming Studio, so as to benefit from the semantic search facilities over all projects files, as well as semantic navigation in the call-graph, def-use references, *etc.*

We reviewed a total of 29 hide blocks, 13 while loops (of which 5 in SPARK code), 10 infinite loops (of which 1 in SPARK code) and 101 exception handlers.

Overall, it took one day to apply code review.

3.3 Categories of Problems

We call *defects* all the problems that may lead to an observable failure of the system to meet the requirements or the specification. We call *code quality issues* all the problems that cannot currently lead to a system failure, but that could lead to system failure during maintenance. *E.g.,* a sorting procedure that does not sort in the appropriate order would be a defect, while a sorting procedure called Remove_Duplicates would be a code quality issue. Note in particular that we considered dead code as code quality issues. Note also that we assume that the formal proof activity is repeated after maintenance changes. We made no effort at assessing the severity of the problems, as the Tokeneer Project is a prototype not meant to be used in reality.

In the following, we primarily present the results in terms of the technique used (whether static analysis or code review) and we distinguish defects and code quality issues found in SPARK code from defects and code quality issues found in Ada code. For some of the issues reported we found multiple instances in the code but we only report them once.

4 Results of Static Analysis

4.1 Defects Found in SPARK Code

Possible Integer Overflow: The first defect in SPARK code was discovered by Rod Chapman, leader of the SPARK team, while preparing the open-source release of the Tokeneer material [6]. The SPARK tools from 2008 are more capable than the same tools were in 2003. In particular, producing proofs of absence of overflow errors could not be justified in terms of time and cost using the 2003 version of the SPARK tools. By 2008 the degree of proof automation had improved so much that arithmetic overflow checking had become feasible, which led to the discovery of one overflow problem.

The summary generated by POGS reported that a VC corresponding to a run-time check at line 233 was undischarged.

```
| 4 | start | rtc check @ 233 | Undischarged
```

It originated in the following lines from configdata.adb, where an overflow could occur during the evaluation of expression (RawDuration * 10) inside the test meant to protect against another overflow.

```
233 if Success and then
234   (RawDuration * 10 <= Integer(DurationT'Last) and
235    RawDuration * 10 >= Integer(DurationT'First))
236 then
237    Value := DurationT(RawDuration * 10);
238 else
```

Wrong Variable Used: Running the GNAT compiler with warnings revealed that a condition in a test was known to be true.

```
keystore.adb:349:23: warning:
condition is always True (see test at line 344)
```

Indeed, the value of some variable RetValIni is tested twice instead of testing the value of another variable RetValDo the second time, which is typical of copy-paste errors.

```
344 if RetValIni = Interface.Ok then
345    Interface.FindObjects
346      (HandleCount   => HandleCount,
347       ObjectHandles => Handles,
348       ReturnValue   => RetValDo);
349    if RetValIni = Interface.Ok then
```

This warning was not present in the version of the compiler used during development of the Tokeneer Project. This defect is also detected by CodePeer in quick mode. Note that it is also detected by the new SPARK Dead Path Analyzer in SPARK 9.0.

4.2 Code Quality Issues Found in SPARK Code

Dead Defensive Code: Running CodePeer in quick mode revealed some dead code, *i.e.,* code which may never be executed in any run.

```
tokenreader.adb:683:19: warning:
dead code because CardState in Absent..Specific
```

It corresponds here to defensive code in the default case of a case-statement. The reason for this code to be dead is quite involved and spans multiple calls.

```
682     when Interface.InvalidCardState =>
683        MarkTokenBad;
684  end case;
```

Global Used instead of Proxy: Running the GNAT compiler with warnings revealed that a parameter in a function was not used.

```
auditlog.adb:585:28: warning:
formal parameter "E" is not referenced
```

Had it been a regular SPARK function, SPARK Examiner would have caught this error immediately, as all function parameters in SPARK must be read at least once. SPARK Examiner could not catch this error because the body of the function was hidden from SPARK tools, using the hide annotation:

```
585  function NameOfType (E : AuditTypes.ElementT )
586    return ElementTextT
587  is
588  --# hide NameOfType;
```

The function's body directly references the global variable ElementID instead of its local name E, most probably because a local block of code was refactored into a local function.

The corresponding warning flag from the compiler was turned off during the development of the Tokeneer Project. Although CodePeer does not issue a warning for this code quality issue, it correctly omits E from the inputs it generates for function NameOfType.

Redundant Conjunct in Precondition: As explained in Section 3.1, we manually translated SPARK contracts into GNAT pragmas, which CodePeer could then understand. Running CodePeer in special mode on the resulting code revealed redundant conjuncts in preconditions, that we report here on the SPARK annotations.

```
enclave.adb:1107:13: warning:
condition is always true
```

The condition mentioned above has the form A → ((B and C) or not B)), where B and C are previous conjuncts in the formula. Since the condition mentioned trivially holds when the previous ones hold, it is useless here.

```
1107  --# (Admin.prf_rolePresent(TheAdmin) = Typ.Guard
1108  --#  ->
1109  --# ((Admin.IsDoingOp(TheAdmin) and
1110  --#    Admin.TheCurrentOp(TheAdmin) = OverrideLock)
1111  --#    or not Admin.IsDoingOp(TheAdmin)));
```

Redundant Conjunct in Postcondition: Running CodePeer in special mode on the translated annotations also revealed redundant conjuncts in postconditions, that we report here on the SPARK annotations for clarity:

```
enclave.adb:1141:13: warning:
condition is always true
enclave.adb:1146:13: warning:
condition is always true
```

The first condition mentioned has the form (A and B) → C and the second has the form D → ((A and B) or not A), where not A is a previous conjunct in the formula. Since the conditions mentioned trivially holds when not A holds, they are useless here.

```
1141  --# ((Admin.IsDoingOp(TheAdmin) and
1142  --#    Admin.TheCurrentOp(TheAdmin) = OverrideLock)
1143  --#  ->
1144  --#    Admin.prf_rolePresent(TheAdmin) = Typ.Guard)
1145  --# and
1146  --# (Admin.prf_rolePresent(TheAdmin) = Typ.Guard
1147  --#  ->
1148  --# ((Admin.IsDoingOp(TheAdmin) and
1149  --#    Admin.TheCurrentOp(TheAdmin) = OverrideLock)
1150  --#    or not Admin.IsDoingOp(TheAdmin)))
```

4.3 Defects Found in Ada Code

Access Out of Array Bounds: Running CodePeer in quick mode revealed a possible access out of an array bounds:

```
tcpip.adb:205:16: medium:
array index check might fail: requires i >= 2
```

Indeed, the second part of the test below accesses Msg.Data at index 0 which is outside of the array bounds, when the first character in the array is ASCII.Lf.

```
204  if Msg.Data(i)     = ASCII.Lf and then
205     Msg.Data(i - 1) = ASCII.Cr then
```

Accessing the Null String: Running CodePeer without partitioning revealed a possible access to the first character of a null string:

```
tokenapi.adb:272:16: medium:
array index check might fail
```

Indeed, MsgProc.GetStringByPos might return a null string, in which case the access to the first character of the result on line 272 is an error.

```
268    HandleStr : String := MsgProc.GetStringByPos
269                              (Msg => Msg, Arg => 1);
270
271 begin
272    if HandleStr(HandleStr'First) = 'p' then
```

Uninitialized Variable: Running CodePeer in quick mode revealed a possible read of an uninitialized value:

```
tcpip.adb:427:20: medium:
validity check: Server might be uninitialized
```

Indeed, local variable Server is read inside an exception handler that might be reached because an exception was raised before Server initialization (code not shown). *E.g.,* this is the case if Create_Selector fails.

4.4 Code Quality Issues Found in Ada Code

Useless Assignment: Running the GNAT compiler with warnings revealed that a local variable was assigned but never used.

```
tcpip.adb:446:07: warning:
variable "Address" is assigned but never read
```

This warning was not present in the version of the compiler used during development of the Tokeneer Project. This code quality issue is also detected by CodePeer in quick mode.

5 Results of Code Review

5.1 Defects Found in SPARK Code

AND instead of OR: The first defect found after the open-source release of the Tokeneer Project was a functional error. It was found by code review by Professor Diomidis Spinellis, an expert in security and code reviews, who reported it on his blog [5].

In the following code from auditlog.adb, the value of AuditSystemFault is cleared if the deletion is successful, and not set if the deletion fails.

```
File.Delete(TheFile => TheFile,
            Success => OK);
AuditSystemFault := AuditSystemFault and not OK;
```

The problem is that a logical “and” operation was used instead of an “or” operation. It could be detected by the SPARK verification tools with precise-enough functional contracts added by a user.

Possible Non-termination: This was detected during our code review focusing on possible non-termination of loops. The following loop in enrolment.adb calls the Ada function File.GetLine to read characters from a file.

```
while Stop=0 and not File.EndOfFile(TheFile) loop
   -- Read the next (non-empty) line of the file
   --# assert PrivateKeyPresent(KeyStore.State) =
   --#        PrivateKeyPresent(KeyStore.State~);
   File.GetLine(TheFile, TheCert, Stop);
end loop;
```

Function GetLine has a catch-all global exception handler, so that if an exception is raised before out-parameter Stop is increased, Stop remains at 0, and the loop may never stop.

5.2 Code Quality Issues Found in SPARK Code

Trivially True Conjunct in Loop Invariant: We found this issue while manually translating SPARK contracts into GNAT pragmas. A loop invariant SPARK annotation in function OpIsAvailable stipulates that KeyedOp maintains its value from one run through the loop to the next run.

```
--#          KeyedOp = KeyedOp%
```

While this is a correct way to specify that the value of a variable modified elsewhere in the function does not change in the loop, it is useless for a constant, which KeyedOp happens to be in this function.

Redundant Conjunct in Precondition: We found this issue while reviewing a warning reported by CodePeer on SPARK contracts.

The code below is part of the precondition of procedure ProgressAdminActivity in enclave.adb. From the first line in ProgressAdminActivity's precondition (not shown here), we know that CurrentAdminActivityPossible holds. Unfolding definitions, we get that

CurrentAdminActivityPossible ↔
  (AdminHasDeparted or AdminActivityInProgress)
AdminHasDeparted ↔
  (not AdminToken.IsPresent and
   Status in NonQuiescentStates)
AdminActivityInProgress ↔
  (Status in ActiveEnclaveStates)

Putting it all together, Status can be in 5 different states: WaitingRemoveAdminTokenFail, GotAdminToken, WaitingStartAdminOp, WaitingFinishAdminOp or ShutDown.

In particular, Status cannot be in the state EnclaveQuiescent, so that the first implication below trivially holds.

The second implication below has the form A → (not B and C) while a previous conjunct in the precondition is equivalent to C → not B, therefore the second implication below could be simply rewritten A → C.

```
--# (Status = EnclaveQuiescent ->
--#  not Admin.IsDoingOp(TheAdmin)) and
--#
--# (Status = Shutdown ->
--#  (not Admin.IsDoingOp(TheAdmin) and
--#   Admin.prf_rolePresent(TheAdmin) =
--#      PrivTypes.UserOnly))
```

The same is true for the postcondition, where these formulas are repeated.

Useless Postcondition: We found this issue while reviewing a low warning reported by CodePeer. This lead us to the following postcondition of procedure StartOp in admin.ads which requires that a predicate is maintained by the call to StartOp.

```
--# post ((Op = OverrideLock <->
--#         prf_rolePresent(TheAdmin~) = Typ.Guard)
--#       ->
--#        (Op = OverrideLock <->
--#         prf_rolePresent(TheAdmin) = Typ.Guard))
```

Looking at the body of StartOp (one line!) immediately reveals that the state upon which this predicate depends is not modified by calling StartOp, which makes this postcondition useless.

5.3 Defects Found in Ada Code

Off-by-one Bug: This was detected during our code review focusing on possible non-termination of loops in msgproc.adb.

```
-- What we found was not the key - delete
-- everything up to this point and search again.
Ada.Strings.Fixed.Delete
  (Source  => LocalMsg,
   From    => 1,
   Through => KeyStart + Key'Length);
```

Here it assumes a quote was found after the string, which is not guaranteed since we are in the else-branch of the if-statement testing precisely this. The value of Through should be instead KeyStart + Key'Length - 1.

Error Code not Set: This was detected during our code review focusing on handling of exceptions in tcpip.adb.

```
   Success := True;
   ...
   -- some code that can raise an exception
   ...
exception
   when E : others =>
      DebugOutput("Send Error.");
```

So procedure SendMsg can return with Success=True while it got a send error. Similar code in other procedures suggests that Success should be set to False in the exception handler. Note in particular that the possibility of an error in SendMsg is correctly taken into account in its caller SendAndReceive.

Ignoring Exceptions: This was detected during our code review focusing on handling of exceptions in file.adb, where PutChar has the following exception handler:

```
exception
  when others => null;
end PutChar;
```

Failures of PutChar will go unnoticed. This is not consistent with the behavior of Copy which calls PutChar and reports Success or not. One level above in the call-graph, ArchiveLog has a different behavior when Copy succeeds or not, so the failure to notify that PutChar failed is altering the behavior of the application.

Ignoring Error Status: This was detected during our code review focusing on handling of exceptions in spark_io.adb, on the following exception handler:

```
exception
   when Status_Error  => Status := Status_Error;
   when Name_Error    => Status := Name_Error;
   when Use_Error     => Status := Use_Error;
   when Device_Error  => Status := Device_Error;
   when Storage_Error => Status := Storage_Error;
   when Program_Error => Status := Program_Error;
end Open;
```

This exception handler translates exceptions into an error status. This is usually bad practice, as this requires all callers of this procedure to check the error status. In crypto.adb, function ClearStore calls SPARK_IO.Open and discards the status variables so that a failure will go unnoticed.

5.4 Code Quality Issues Found in Ada Code

Possible Access Out of Array Bounds: This was detected during our code review focusing on possible non-termination of loops in msgproc.adb.

```
-- Found the Key string- is it enclosed by quotes?
if LocalMsg(KeyStart - 1) = ''' and
   LocalMsg(KeyStart + Key'Length) = ''' then
```

There is no guarantee that KeyStart - 1 and KeyStart + Key'Length are within bounds. The catch-all exception handler in the function correctly returns a default value in this case, but is certainly not adequate for high-integrity software to raise a Constraint_Error, even locally.

Dead Subprograms: Procedure Compare and function Get_Name in file.adb are syntactically dead subprograms (never called).

Implicit Invariant: This was detected during our code review focusing on handling of exceptions in certproc.adb.

```
SigDictStart := Ada.Strings.Fixed.Index(
   String(CertData), String(SigDict));
```

If two keys have same dictionary, the call to Index will return the index of the first dictionary found, not necessarily the one that is associated to the key. Then, the subsequent call to Delete will fail to delete the proper entries. Therefore, the functional behavior of the program depends here on the implicit invariant that no two keys share the same dictionary, which is not expressed anywhere in code or comments.

6 Summary of the Results

Tokeneer	# SPARK defects	# SPARK issues	# Ada defects	# Ada issues	# total (# exclusive)
GNAT	1	1		1	3 (0)
CodePeer	1	4	3	1	9 (6)
SPARK tools	2				2 (1)
static analysis	2	4	3	1	10
hide annotations					0
loops	1	1	1	1	4
exceptions			3	1	4
other	1	2		1	4
code review	2	3	4	3	12
total	4	7	7	4	22

Table 1: Defects and issues found in Tokeneer

Table 1 summarizes our findings. No quantitative conclusions should be made based on the figures reported here:

- Concerning the suitability of this or that approach to find problems, the code being analyzed had already been cleared by the GNAT compiler, SPARK Examiner, and SPARK Simplifier during development, while this work was the first application of CodePeer, SPARK Dead Path Analyzer, and the various focused code reviews on this code;
- Concerning the comparative density of problems found in SPARK and Ada, we mostly concentrated during this work on finding problems in SPARK code, both when reviewing static analysis results and when performing our focused code reviews.

The Summary Report for Tokeneer [4] contains detailed quantitative information about the problems found during the project vs. point of introduction.

In Table 1, the number of defects (or issues) for each column found by static analysis may be less than the sum of the problems found by individual tools, as the same problem may have been found by more than one tool. For each tool used, we detail in parentheses the total number of problems that this tool is the only one to discover. In the following, we draw qualitative conclusions based on these figures and additional information reported in Sections 4 and 5.

The fact that 4 defects were found in SPARK code, despite considerable efforts spent in the assurance process during the Tokeneer Project, is a clear indication that use of formal verification does not guarantee that the system built will be free from defects. Looking at the defects found, only one is a run-time error (possible integer overflow) and it was found with a new version of the SPARK tools. The possible non-termination clearly points to a "blind spot" of the SPARK verifications, which can be addressed on the user side by annotating possibly non-terminating loops with loop variants (currently encoded as loop invariants). A way to express loop variants in SPARK annotations would certainly be a much better solution here. The last two defects found in SPARK are functional errors, which would be detected automatically if the code contained enough functional annotations expressed as SPARK contracts. Here, the limiting factor is the cost/benefit ratio of adding such annotations, which lead to the decision to only develop security annotations in the Tokeneer Project. Interestingly, one of these functional errors (wrong variable used) is now found by all the tools as a side-effect of checking for conditions always true.

In comparison, the defects found in Ada code are much more serious. The defects found by static analysis are all serious run-time errors: access out of array bounds, accessing the null string, uninitialized variable. The defects found by code review are all serious functional error: off-by-one bug, error code not set, ignoring exceptions, ignoring error status. Most of these defects cannot be present in SPARK code by construction, as they will be caught by either SPARK Examiner or SPARK Simplifier. This is a clear indication that use of formal verification does guarantee that the system built will be free from a large set of serious defects.

Most of the code quality issues found in SPARK code have to do with lack of coverage in code (dead code) or in annotations (redundant annotations). Even if static analysis and code review uncovered a number of these, this is an area where dynamic coverage analysis is needed to guarantee some degree of coverage. *E.g.,* dead code would be found by doing structural coverage and redundant annotations by doing decision coverage or MC/DC coverage. This would have caught also the dead subprograms found in Ada.

It is interesting to note that one code quality issue (global used instead of proxy) could have been detected by running the compiler with the appropriate warning switch. This is another example that tool setup and configuration is as critical as tool choice. Tool configuration needs to be reviewed, configured, and fault-managed even more thoroughly than code, as this one review (of the tool configuration - *e.g.,* compiler switches) may very efficiently replace a much bigger code review effort.

A crucial point is that most of the time spent in these experiments, both for static analysis and code review, was some kind of manual review: review of the tools output for static analysis, review of the code itself for code review. An equally crucial point is that tool support for these reviews was very important. The ability to search and navigate the semantic structures of the code in an IDE allows the user to focus on understanding the warnings and the code itself, without getting distracted by the details of repeatedly finding appropriate source files or the elements in the source files.

As expected, most time spent during the review of static analysis results consisted in identifying true problems from false alarms. An interesting point not widely recognized is that code reviews also produce false alarms: we spent much more time on two false alarms produced this way than on any false alarm produced by static analysis tools. In the first case, we thought a Prolog user-defined rule for SPARK Simplifier was wrong, while it was not; in the second case, we wrongly identified dead code by mistaking File.EndOfLine(TheFile) for File.EndOfFile(TheFile).

As we focused mostly on the SPARK code, we did not make much use of the contracts generated by CodePeer, which make explicit the otherwise implicit inputs, outputs, preconditions and postconditions of subprograms. Had we focused more on Ada code, we would certainly have relied more on the contracts generated by CodePeer.

7 Associating Static and Dynamic Analyses

Considering more broadly the industrial practice in verifying properties of software, it is currently mostly supported by applying a process of development (including coding rules and code review) and by dynamic analyses (including testing and coverage analysis). In parallel, static analysis techniques for the sake of finding problems have reached a degree of industrialization which allows their use on very large systems. These techniques generate warnings but in general they give no guarantees (although CodePeer does so) and the treatment of false alarms requires a costly expertise (which is why we ignored low warnings from CodePeer in our experiments). Only a few industries have turned to static analysis for the sake of formal verification [8], using *e.g.,* SPARK.

We propose to associate static analyses and dynamic analyses with the aim to achieve better software verification. The goal of this association is both to complete the formal verification efforts with different static analyses (like CodePeer) and dynamic analyses, and to popularize formal verification for those who already use code reviews, testing or static analysis.

To that end, we propose to base all analyses on the code itself, thanks to the extensions of the Ada language with contracts proposed by the Ada Rapporteur Group for the next version of Ada, Ada 2012. For example, we would express the contract of function Sqrt which computes the square root of its argument as follows:

```
function Sqrt (X : Integer) return Integer
  with
    pre  => X >= 0,
    post => Sqrt'Result * Sqrt'Result <= X and
       X < (Sqrt'Result + 1) * (Sqrt'Result + 1),
    test => if X = 10 then Sqrt'Result = 3;
```

Notice in particular the expression of a test case in the contract of Sqrt.

Annotations give us the possibility to group together the specification, the tests and the code in the same place, which solves by construction the problem of traceability between these artifacts of the development process. Tests will allow annotations to be easily tuned by execution and debugging. Some static analyzers may generate annotations in order to facilitate formal verification, much like CodePeer already generates subprogram contracts to facilitate code review.

Our goal is to provide “lite” tools and methods for the development of high integrity applications, that developers may apply to ongoing implementations on their computers to verify some safety properties and logic properties. Long after the first version of the popular software Lint (1977) to find programming errors in C programs, we propose to build a High-Integrity Lint.

Together with our focus on testing and execution, this gives the name of our project: Hi-Lite for High-Integrity Lint Integrated with Testing and Execution. All tools in this platform will be free software. Sources for all new tools will be available through the Open-DO initiative (`http://www.open-do.org/`).

8 Conclusion

In the introduction, we listed several different reasons for a formally verified system to contain problems. In this paper, we provide empirical evidence that such problems show up in practice: the presence of code that is not verified formally; the imprecision of user specifications; the inability of formal verification techniques to target some properties. In Section 2 we presented the Tokeneer Project, which was formally verified to the higher assurance levels of the Common Criteria. In Section 6, we discussed 11 defects and 11 code quality issues found in both SPARK (formally verified) and Ada (not formally verified) source code of the Tokeneer Project. Only two of these were discovered before this work. All other defects and code quality issues were found in 3 days of work by the application of static analysis tools and code review.

In related work [7], Woodcock and Aydal have found security errors in the specification of the Tokeneer Project. They translated the Z specification into Alloy notation and analyzed it using the Alloy analyzer, a bounded model checker based on SAT solving. They discovered 12 mild but actual problems, of which they give a detailed account for 4 of them.

The results of both studies indicate that formal developments would benefit from being integrated in a larger tooled approach. The number of problems highlighted should not cast any doubt on the fact that this number was not much larger only because the Tokeneer Project was developed using formal methods. It is a remarkable achievement that this complete redevelopment of an actual system by 3 people over 9 months (part-time) achieved such a level of quality that only a few problems can be found several years later using so many new approaches not available at the time.

In order to complete formal verification efforts, we propose to associate static analyses with dynamic analyses in a new project called Hi-Lite. Hi-Lite's goal is to promote the use of formal methods in developing high-integrity software. It loosely integrates formal proofs with testing and static analysis, thus allowing developers to combine different techniques around a common expression of properties and constraints. Hi-Lite's focus on modularity allows a divide-and-conquer approach to large software systems and encourages early adoption by all programmers.

9 Acknowledgment

We are grateful to Rod Chapman and Diomidis Spinellis who inspired this work by finding the first two problems on Tokeneer by static analysis and code review, and to Arnaud Charlet who found more problems by running CodePeer for the first time on Tokeneer's code. We would also like to thank Neil White for his useful remarks on the manuscript.

10 References

[1] Janet Barnes, Rod Chapman, Randy Johnson, James Widmaier, David Cooper, and Bill Everett. Engineering the Tokeneer enclave protection software. In *1st IEEE International Symposium on Secure Software Engineering*, March 2006.

[2] Anthony Hall and Roderick Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(1):18–25, 2002.

[3] ISO/IEC Standard 15408. *Common Criteria for Information Technology Security Evaluation*, version 2.1 edition, August 1999.

[4] NSA and Praxis High Integrity Systems. EAL5 demonstrator: Summary report, August 2008. `http://www.adacore.com/multimedia/tokeneer/Tokeneer_Report.pdf`.

[5] Diomidis Spinellis. A look at zero-defect code. *Personal blog*, 2008. `http://www.spinellis.gr/blog/20081018/index.html`.

[6] `http://libre.adacore.com/home/products/sparkpro/tokeneer/downloads/`.

[7] Jim Woodcock and Emine Gökçe Aydal. A token experiment. *Festschrifts in Computer Science, the BCS FAC Series*, Festschrift for Tony Hoare, 2009.

[8] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):1–36, 2009.

11 Glossary

Hi-Lite:	High-Integrity Lint Integrated with Testing and Execution
IDE:	Integrated Development Environment
kloc:	thousand lines of code
MC/DC:	Modified Condition / Decision Coverage
NSA:	National Security Agency
POGS:	Proof ObliGation Summarizer
VC:	Verification Condition