# Integrity Static Analysis of COTS/SOUP

Peter Bishop[12], Robin Bloomfield[12], Tim Clement[2], Sofia Guerra[2] and Claire Jones[2]

CSR, City University[1] Adelard[2]
Drysdale Building, Northampton Square London EC1V 0HB, UK
mailto:{pgb,reb,tpc,aslg,ccmj}@adelard.com

**Abstract.** This paper describes *the integrity static analysis* approach developed to support the justification of commercial off-the-shelf software (COTS) used in a safety-related system. The static analysis was part of an overall software qualification programme, which also included the work reported in our paper presented at Safecomp 2002 [1]. Integrity static analysis focuses on unsafe language constructs and "covert" flows, where one thread can affect the data or control flow of another thread. The analysis addressed two main aspects: the internal integrity of the code (especially for the more critical functions), and the intra-component integrity, checking for covert channels. The analysis process was supported by an aggregation of tools, combined and engineered to support the checks done and to scale as necessary. Integrity static analysis is feasible for industrial scale software, did not require unreasonable resources and we provide data that illustrates its contribution to the software qualification programme.

## 1 Introduction

This paper describes the integrity static analysis approach that was developed to support the justification of the use of a Commercial-Off-The-Shelf (COTS) industrial product in a safety-related application. This paper is a continuation of the work presented at Safecomp 2002 [1]. Integrity static analysis focuses on unsafe language constructs and "covert" flows, where one thread can affect the data or control flow of another thread. The system to be analysed was a general purpose C&I system that had been used in control and protection applications for over 10 years and was going to be deployed in a new safety application.

The static analysis was part of an overall software qualification programme motivated by the need to demonstrate confidence in the safety of the system. The primary motivation of the programme was not to find faults in the software but to increase the confidence that it was appropriate for the new safety application. The qualification programme involved several activities, of which static analysis was only one. In addition to static analysis, it also included:

- Evaluation of the operating history and review of the hardware and software problems detected after product release and during operation.
- Evaluation of the design and lifecycle documentation.
- Identification of the supporting tools used, and evaluation of their use and criticality (including compiler assessment).
- Software criticality analysis, as described in [1].

- Dynamic testing and especially stress testing.

The primary task of the static analysis was to check the structural integrity of the code and hence provide additional evidence of the software quality. This examined two main aspects:

1. The internal integrity of the code (especially for the more critical functions).
2. The intra-component integrity, checking for "covert channels" where one software component could affect another by some "back-door" method.

In [1] we described the Software Criticality Analysis (SCA) approach. The SCA identified the critical software components within the COTS software, and this information was used to prioritise the safety justification activities for the whole project. Analysis effort was concentrated where a fault would be more dangerous, i.e. the components with higher software criticality indices, as assigned by the SCA.

However, the SCA considered only overt flows in the code, via the call structure of the procedures and the way in which static variables are shared amongst them. Where many procedures share a single processor and address space, there are potentially other covert ways in which one may influence another, and evidence was needed that these covert channels were not present in practice to justify use of the SCA. This was addressed by the integrity static analysis described in this paper, and it was another objective for the approach described here.

**Code description.** The code comprised a real-time PLC operating system running on top of a commercial microkernel. It included device drivers for various different types of hardware, scheduling, message handling and tasking. It consisted of a large body of mixed C and assembler code totalling about 600 files, and 100k+ lines (non-comment, non-blank) of C and 20k+ lines of 68000 assembler. The C code had two components: one of about 100k lines and a second part of about 10k lines. Similarly, the assembler code had two components. However, while the two C components differ in size and style (the smaller component had been developed more recently and was more consistent in style), the two assembler components were almost identical.

The code had extensive field experience. It had been sold to users in various versions over a number of years. As expected, the code contained some very old modules that had been in use in many sites for many thousands of operating hours, and some modules which had recently been changed and which had little or no operating history. Parts of the code had been automatically translated from a different language to C. The main reason for changes had been adding new software, e.g. for new hardware, and maintenance (e.g. fixing existing bugs).

## 2 Integrity static analysis

### 2.1 Motivation and objectives

The analysis approach adopted was motivated by two main factors. Firstly, the COTS system had extensive field experience, and the field data was being analysed as part of the overall qualification exercise. The field experience is likely to detect most large and obvious faults that occur during typical execution of the program. However, specific vulnerabilities of the languages used and the particular domain of application have a less frequent manifestation that could remain undetected even after many hours of field experience.

Secondly, the analyses that could be performed were constrained by feasibility, resources available and tool capabilities. For example, for a compliance analysis such as the work done with MALPAS in Sizewell B PPS [2], we would need to develop and verify a formal specification for a large body of heterogeneous C and assembler code and follow this by a proof of compliance using a tool like MALPAS. This would be very time-consuming, and some features such as the use of pointers in the COTS software make it very difficult to prove compliance. Even if a proof of compliance were achieved, analysis using MALPAS would not provide complete assurance as the analysis can only be applied to sequential, non-interruptible code. The COTS software contained many concurrent threads that could modify data in other threads and hence potentially invalidate the proof.

These resource and technical constraints led to the decision to perform an *integrity static analysis* that focuses the analysis on unsafe language constructs, and "covert" flows where one concurrent thread can affect the data or control flow of another thread. The main aspects analysed are listed below (further details in Section 4).

**Table 1.** Main aspects analysed in the integrity static analysis

| **Unsafe Language Constructs** | **Covert flows** |
|---|---|
| Function prototype declaration missing. | Resource sharing violations (semaphores, interrupts, etc.). |
| Use of "=" in conditional expressions. | Violation of program stack and register constraints (in assembler code). |
| No return value defined for a non-void function. | Pointer or array access outside the intended data structure. |
| No break between case statements. | Run-time exceptions (e.g. divide by zero). |
| Uninitialised variables (Possibly but not necessarily covert flows). | |

The assessment of unsafe language constructs identifies potential vulnerabilities in the C code by looking for deviations from published recommendations for C programming in safety-related applications [3][4] and use of features of C identified in the ISO and ANSI standards as ill-defined or dangerous. It also includes checks for a variety of specific issues, such as the use of commonly misused constructs in C (such as "=" in conditional expressions).

Covert flow analysis examines the potential interference between different code functions. The most obvious covert mechanism in C or assembler code is the use of pointers (including their implicit use in array indexing). An incorrectly calculated pointer to a variable can give a procedure access to anywhere in the program's address space. Similarly, incorrect pointers to functions allow transfer of control to anywhere in the address space. The sharing of resources on a single processor and sharing of the stack give rise to other covert mechanisms. Static analysis was used to support an argument of the absence of these covert channels.

We observe that the existence of these covert mechanisms represents an error in the code, irrespective of its intended function, i.e. "whatever it is meant to do, it should not do this". We can thus carry out these analyses without a formal specification of the intended function, which is fortunate since this was not readily available. A written specification would have to be formal to support an automatic analysis, something we do not expect for legacy code, and domain experts were in

short supply so code anomalies needed to be assessed where possible without the use of experts.

Where the static analysis uncovered significant faults, the code was corrected. Even within the scope of the analysis, limitations in supporting tools and manual analysis may mean that not all the faults sought will be discovered. We cannot therefore claim that the resulting code is fault free, and there is no compelling reason to repeat the analysis on the revised code provided the revisions are done carefully. However, if the static analysis detects relatively few faults, this does help to boost confidence in the code as a whole.

### 2.2 Analysis process

The overall procedure used in the static analysis tasks was as follows:

- *Identification of preliminary findings.* Preliminary findings were identified as a result of tool analysis. In some cases (especially uninitialised variables), these were further processed by automatic filters that removed some of the more obvious cases where there was no real problem. These filters were designed to be conservative so any findings about which doubt remained were passed through to the next stage.
- *Provisional sentencing.* Manual inspection of the preliminary findings assigned a provisional sentence. This preliminary sentencing was based on code inspection with limited domain knowledge. Where it did not seem possible to sentence the finding without extensive domain knowledge, an open sentence was recorded together with any information that could help the sentencing.
- *Domain expert sentencing.* All the findings with a provisional sentence other than “no problem” were reported to the client. The majority was resolved and justified. A small residue of findings was accepted as genuine and sentenced for severity and technical solution.
- *Review of the final sentencing.* The analysis team reviewed the domain experts’ sentencing. Clarification of the final sentence was asked for in the cases where the justification was not clear. The final decision on the solution to adopt was taken by the domain experts.

## 3 Tool support

This section describes the tools used to support the analysis. The main tools are described according to their main features and use in the project, followed by a brief assessment of our experience with the tools. In addition, we name other general purpose tools used in the analysis and discuss tool evaluation issues.

### 3.1 PolySpace

The RTE tool from PolySpace [5] seeks to identify those points in a C program that will cause a run-time exception. These include both exceptions due to arithmetic

overflow and the like, identified above as covert channels affecting control flow, and erroneous pointer and array accesses, which may result in exceptions if the address is outside physical memory. Unlike the other sources of covert flows in C tied to particular statements, run-time exceptions can arise at many points in the code and depend on the variable values at the time of program execution. This makes their analysis relatively difficult and so the use of a tool to carry out most of the work is particularly attractive.

RTE uses abstract interpretation [8] to analyse the code. It attempts to show that each potential run time exception cannot occur by establishing the possible values of the expressions involved. This requires analysis to determine the possible values of variables at each point in the code.

When the tool classifies a variable usage as not causing an exception for any value in the predicted range or as causing an exception for all possible values, the analysis is completed for that particular occurrence. However, if only some of the possible values can cause an error (marked as orange), because of the approximations, more detailed analysis of the code might sometimes show that the error could never occur. The power of the tool depends on the fraction of all possible exceptions that can be automatically labelled either red or green. PolySpace refers to this as the selectivity ratio and reported typical values on large programs in the range 85-95%.

In order to trace the flow of values through the program, PolySpace RTE requires C source for all procedures except for those in the C standard library. For the system under investigation, this meant “stubs” had to be created to represent the procedures provided by the microkernel and for assembly language routines. Tracing value flow also requires that the semantics of the code should be well defined, which implies strict adherence to the ANSI syntax. For the purpose of the analysis, the code had to be changed to be ANSI compliant, where the majority of changes involved the removal of type information from bit fields in structures.

The primary problem in using the tool was its scalability. Although PolySpace had analysed other programs of similar size, the larger component of our code defeated the analysis. This may have been a consequence of the relatively complex task structure of the system. The smaller component (10k loc) was successfully analysed, but the selectivity rate was only 50%. We also identified a number of cases where manual analysis showed the tool to be overly cautious. We understand that better results have been achieved elsewhere, and the analyser has undergone considerable development, so we would expect experience with the new version to be better.

### 3.2 CodeSurfer

CodeSurfer [6] is a Grammatech tool that analyses C programs to construct graphs showing the data and control flows through the program. The user can then explore the graphs to obtain an understanding of connections between different program parts. The analysis and exploration phases are separate, with the analysis phase producing an intermediate file containing a *system dependence graph* (SDG) that the browser program then uses to present different views of the program. Program views vary from a list of program lines where a particular variable is used, to a *slice* highlighting the program lines that can affect the value of a variable at a given point in the code (a *backward slice*) or are affected by it (a *forward slice*). CodeSurfer also provides an applications programmers’ interface (API) which allows the user to write scripts that

explore the program graphs and display the results in windows or write them to a file. CodeSurfer traces data flows through pointers by determining the set of variable addresses that each pointer may be assigned anywhere in the code, either directly using the & operator or indirectly by copying addresses from one pointer to another.

We originally used CodeSurfer to extract call graph information for the SCA [1]. Here CodeSurfer was used to support the analysis of variable initialisation.

CodeSurfer expects its input to be ANSI C but is less strict about full compliance than PolySpace RTE. A number of minor changes had to be made to the supplied code. The larger component of the software contained too much code for analysis as a single "project", the limit being determined by the size of the system dependence graph rather than the analysis time needed. We therefore had to divide the code into several projects, analysing them separately and combining the results in a database.

We uncovered a bug in the code flow analysis that left some of the conclusions in doubt. At the time, although Grammatech were supportive in advising us on how the tool could be used, we had some difficulties in obtaining information about potential problems. There is now an on-line mechanism for reporting problems and suggestions, and a public list of known issues with the tool.

While CodeSurfer has some limitations, it proved to be a useful tool for static analysis. The scripting mechanism was vital for extracting information from projects, combining it to give a view of the whole system, and representing it in a form that could be used to support further analysis.

### 3.3 Safer C

The primary aim of the Safer C tool is to check compliance of C code with a set of almost 700 rules. These cover the dangerous constructs identified in [7], the features of C identified in the ISO and ANSI standards as ill-defined or dangerous, those MISRA C requirements and guidelines [4] that can be checked mechanically, and other potential flaws in C code that can be identified by file-by-file analysis with limited tracking of control and data flow. As well as this rule based checking, Safer C can derive some code metrics. Finally, it is possible to select a variable and display its declaration, or do data-flow analysis by highlighting its uses elsewhere in the code.

We used Safer C in the analysis of pointers and arrays, and uninitialised variables. Once it had been adopted, it made sense to make use of the information provided on the occurrence of suspicious C constructs (such as the use of "=" rather than "==" in conditional expressions), so this was added to the approach.

Our approach to pointer analysis was to use Safer C to identify all relevant pointer uses in the code and to examine each of these by hand. Array access calculations were not counted as pointer arithmetic in Safer C, but Oakwood Computing added this rule for us, and the validity of the array indexing was checked manually.

Safer C adopts the strict ANSI view that static variables are initialised when the program is loaded, so all the reported uninitialised variables are local. We covered the explicit initialisation of static variables with CodeSurfer alone. In addition, Safer C was modified by Oakwood Computing at our request to flag division by non-constant values (to confirm by code inspection that division by zero would not occur).

Safer C actually consists of a series of analysis and browsing programs that communicate via an interface that is normally controlled from the visual front end.

Oakwood Computing provided documentation and guidance on this interface so that we could run the component tools on a file in batch mode from a DOS command line and extract the results from the files created. The command lines to analyse each file in our code body were generated by a `perl` script.

Safer C accepts a superset of the ANSI C language definition. However, a small number of code changes was needed to allow the tool to correctly parse and interpret the files. Because Safer C analyses each file in the source separately, we had no problems in applying it to the entire body of C code.

The tool makes no claims for the completeness of its uninitialised variable analysis, but it is not entirely clear what the limits are. When we compared the cases identified by Safer C and by our CodeSurfer analysis on real code, we found there were not many cases that Safer C had missed but CodeSurfer had found. We also ran the Safer C tool on the smaller part of the code and compared its results in detecting potentially uninitialised variables with the RTE results. The sets of possibly uninitialised variables identified by RTE and Safer C were different, but both included the few cases that were accepted as a problem and resulted in software modifications. A new version of Safer C works on a collection of files for checking global consistency (rather than only individual files, as the version used here).

### 3.4 Other tools

The previous sections summarise the capabilities and uses made of the major, special purpose tools that we employed. We should not lose sight of other, widely spread tools that played a significant role in the analysis.

Assembler code made up a significant part of the code body. We needed to subject this to analysis of its pointer and stack use, but there was no special tool support available for this. Instead, we followed the strategy of mechanically marking all those lines that could cause a problem and considering each in turn to determine if it did. Assembler code is relatively simple in structure, and we used `perl` and `grep` to extract the lines of interest. This approach is less effective in C because the language is more complex (although it was used for identifying resource locking statements).

These string manipulation languages were also invaluable in translating from the different textual output forms of the various tools into tab or comma separated, columnar data files that could be read into Microsoft Access database tables.

Access also filled a critical need for tool support. Many analyses involved the identification of large sets of possible problems that were then resolved by hand. This manual analysis was supported by Access. Access was also used to merge findings from different sources and in mechanical sentencing of findings. Some of analysis was programmed using Visual Basic for Applications within Access.

### 3.5 Tool evaluation issues

Despite efforts to ensure the C analysis tools chosen would meet our requirements, we had significant difficulties in applying them to the COTS software. The most serious of these was scalability. All tools had difficulties coping with dialects of C, which were inconsistent between tools and needed code changes. All tools failed to identify

some instances of anomalies they were designed to detect. A standard set of "benchmark programs" would be valuable in providing a basis for evaluation of static analysis tools prior to their use on a project. The benchmarks could cover such issues:

- limits in code size
- analysis time (and how it varies with size)
- analysis resources (memory and file storage-—and how they vary with size)
- reliability in detecting anomalies (for different types of anomaly)
- "signal to noise ratio" (how many anomalies reported with a single cause)
- likelihood of "false positives" (anomaly reported where none exists)

## 4 Activities of the integrity static analysis

This section describes the analysis of the C and the assembler code by discussing the main activities of the integrity static analysis, their objectives and techniques.

### 4.1 Resource locking

This analysis aimed at identifying all points in the code where some resource was claimed and checking that the resource was released (without excessive delay) on all possible program paths through the code. This prevents cases where a low criticality section claimed some "resource" that was never subsequently released, which could prevent a critical code section from being able to access this resource and cause a software failure. As a secondary effect, this analysis demonstrated the absence of failures due to exhaustion of resources, and some types of deadlock or livelock due to unreleased resources. The analysis process may be summarised as follows:

- Identify possible occurrences of "locking".
- For each one, search for a corresponding "unlocking" on all paths.

The domain experts identified a set of code statements as defining the locking or unlocking of a resource. We assumed that this list was complete and included all locking and unlocking statements and consequently all resource locking constructs.

In the case of C code, resource locking statements were identified using a textual search. In order to identify the corresponding unlocking statement, the neighbouring code for each "locking" code statement was examined. At this stage, we examined the branching structure of the code, as in some instances multiple unlocking is required. Further examination of the code was performed in order to validate the analysis.

In the C code, there were often many lines between disabling and enabling interrupts. This made it impossible to judge is enabling was taking place within a short space of time without domain knowledge. We provided the domain experts with a list of matched instances of interrupts disabled and enabled, ordered by the number of lines of code in the disabled section. This list could be used for further review.

For the assembler code, we began by cutting down the amount of code that needed to be reviewed by using a `perl` script. The script identified and annotated the sections of the code that contained the resource locking statements. The extraction process removed comments and irrelevant code. The remaining code displayed locking and unlocking statements, calls to subroutines, program branches and operations that

affect the stack. In more complex cases there may be splits in the program flow before the resource is unlocked. In this case each flow path had to be inspected to check that the locked resource had a matching release statement. In addition any calls to routines within the locking statements needed to be checked to ensure that they did not introduce excessive delay, and that program flow always returned the calling routine.

### 4.2 Stack allocation

The stack is a global resource for a task, shared by all procedures. The stack pointer is initialised and space for the stack is allocated at system start-up. It then grows and shrinks as procedures are entered and exited. If the wrong variables are addressed, a wide range of erroneous behaviours could occur. For example, if the stack of an adjacent task were overwritten, it might execute arbitrary areas of code or data in the memory with extremely unpredictable results, usually resulting in a rapid crash.

The C source code was assumed to be free of stack allocation problems as stack frames and variable references are generated by the compiler, and correct use of the stacked procedure arguments is enforced by function prototypes. The focus of the stack analysis was therefore the assembler code, although we also considered cases where C and assembler code routines interact.

The first objective of the analysis was to assess stack integrity for assembler procedures to ensure the data region accessed by the stack always remained within bounds. The second objective was to check that register integrity was preserved when C calls were made from assembler code. The C compiler guarantees that some registers' values are preserved after the call, but the remaining registers could be destroyed. The register integrity check therefore confirmed that the vulnerable registers were saved on the stack prior to a C routine call and correctly restored afterwards, or that the registers changed by the C call did not affect subsequent assembler behaviour.

In order to assess stack integrity, the assembler sources were scanned using a `perl` script to identify assembler files that contain stack manipulation instructions, i.e. those that reference the stack pointer register explicitly and implicitly. A reduced assembler program was produced (similar to that produced for the resource locking analysis) that highlighted the stack manipulation instructions. The reduced assembler was inspected to check that the stack integrity criteria were satisfied.

A similar approach was used for the register integrity analysis where assembler code calls a C routine. C routine calls were all identified by a label.<procedure_name>. A simple script was used to identify the assembler files that call C routines and the code sections before and after the call. These code sections were analysed manually to check that the registers were correctly saved and restored.

In cases where a C routine was called and there was no stacking of registers, the code was inspected to check if the potentially corrupted registers were relied upon after the C call. There was a difficulty in checking whether any parents in the assembler call tree were also affected. However, in some cases it could be demonstrated there were no parents (e.g. it handed control back to the microkernel).

### 4.3 Pointer and array use

In this system, all the code shares a single address space. This means that all code has access to any data structure if it generates the appropriate address. This can happen as a result of a fault in an address calculation, which results in the generation of an address beyond the data areas that the code apparently accesses. As well as pointer dereferences, this includes access to arrays determined by index calculations.

"Wild" reads will have an impact only on the procedure generating the read, and so do not affect the assessment of criticality. "Wild" writes, on the other hand, allow apparently low criticality code to affect high criticality code. Hence, when an assignment to a pointer occurs, we need to know that the procedure doing the assignment is expected to access the address to which the pointer is being assigned. This means tracing back through the calculation of the pointer value. Pointers are often initialised with the address of a variable. Subsequent pointer references are valid and so are unproblematic. The most common way to create a wild pointer is through a faulty address calculation. Manipulations to pointers also include "casting" to a pointer of a different kind. This may allow access to adjacent data storage if the resulting type is larger than the original type. Subsequent address calculations will also have a different element size that could lead to the calculation of an out of range reference even if the pointer is subsequently cast back to a correctly sized object.

The analysis of the assembler code started by the identification of the code to analyse. A `perl` script was used to parse the code into label, operator and operand fields. Once we identified the lines where calculated addresses are used for writing, we associated each with a claim that the procedure is allowed to write to the address of the instruction. We then manually propagated the claim back through the previous instructions. The propagation stops when the truth of the claim can be determined. This is typically when we have a constant address. Alternatively, we might reach a label that has no transfers of control to it from the assembler code. These labels are normally defined as external symbols that can be called from C. In principle, these need to be considered call by call to check that appropriate pointer values are being passed from the C code. In most cases, the function prototype would show that the address of a structure was being passed, and it was possible to show that the offsets from the base address being used by the assembler code lay within the structure. Provided all calls were in the scope of the function prototype (which was checked by the Safer C analysis) we could then be sure that the pointer use was safe.

The analysis of the C code was divided in two parts. The smaller part of the code (~10kloc) was supported by PolySpace RTE, and the main part by Safer C.

The PolySpace tool categorises potential sources of pointer errors in the code (and run-time errors in general) as green, orange or red, with red representing certain failure points and orange representing possible failure points. The objective of this analysis was to look in detail into the red and orange sections of code, resolve the red findings if any, and establish that the orange sections do not result in failures. The code was then examined to find a justification for the absence of error. (The process was similar for all RTE supported checks).

The analysis of the main part of the C code was supported by the Safer C tool. In addition to pointer arithmetic, the analysis also considered assignment of a structure or structure component to a pointer that might have been initially allocated to a

smaller structure (widening and narrowing). In addition, we looked at all array accesses to check that the index variable was within the bounds of the array.

### 4.4 Uninitialised variables

Uninitialised variables are a well-established source of software failures. Although their most direct effects are on the procedures which use them, their consequences can cross the boundaries between software of different criticalities, by causing run-time exceptions (through numeric overflow or underflow, or by generating an address that is not present in the hardware) or by direct corruption of another procedure by writing through a pointer derived using an uninitialised value.

Local variables are allocated on the stack in C and will contain whatever was previously placed in their stack frame until explicitly assigned. The C standard requires global and file static variables to be cleared by the compiler, but a null value is not necessarily appropriate and there is no guarantee that they will be re-initialised if the system is restarted, so we expect an explicit assignment to these variables too. We therefore checked that all global variables were explicitly initialised. The analysis of the smaller part of the code was supported by RTE. The main part of the code was analysed using a combination of Safer C and CodeSurfer. Both are capable of identifying uninitialised local variables, although the extent of the analysis is different in each case. Each tool led to a large number of findings, which were followed up by a range of mechanical and manual analyses to find the actual problems.

## 5 Analysis results

### 5.1 Summary of static analysis results

The integrity static analysis involved a combination of tool based and manual analysis that examined 100 000 lines of C and 20 000 lines of assembler code for typical vulnerabilities of real-time software. In this analysis, all the C code that was active (or used) in the application was analysed (around 70% of the main part and 100% of the smaller part). The unused code was justified as not interfering as a result of the SCA [1]. The analysis effort was ~3 person days/kloc.

The performance of this process is summarised in the Table 2, where “preliminary” are the findings identified by the tools, “reported” those reported to the domain experts, and “sentenced” those sentenced as other than “no problem”.

**Table 2.** Number of finding per kilo line of code

| Finding | Approx. findings per kilo line of code |
|---|---|
| Preliminary | 100 |
| Reported | 10 |
| Sentenced | 1 |

The sentenced findings were classified as follows (in order of increasing severity):

- 70% were found to be “minor violations”. These have no impact on safety or on system operation but are a violation of good practice.
- 13% were found to be “significant violations” where the code is safe in its current version (cannot cause a run-time problem), but represents a violation of good practice or a potential fault, which could be activated by maintenance.
- 17% were classified as “minor safety”. All minor safety issues should be addressed through code changes in the software.

In Table 3 we compare the results of the static analysis with the other sources of evidence undertaken in the overall validation programme.

The additional analysis consisted of evaluation of lifecycle documents, evaluation of supporting tools and dynamic testing, especially stress testing. From the results, we can see that our static analysis revealed a majority of the findings (55.6% of the total findings). On the other hand, the analysis does not necessarily find the most safety-critical faults: it only found a third of the “minor safety” faults. Most of the major faults were found by field experience prior to the static analyses being undertaken.

The separate field experience analysis indicates that only a small percentage of field-detected faults are simple logic errors (i.e. detectable by static analysis of code structure). The majority of faults were due to:

- hardware/software mismatch (e.g. failure to modify the software for new hardware)
- timing problems (e.g. incorrect timing, incorrect time-outs etc.)
- protocol problems (e.g. between processors interface devices, etc.)

**Table 3.** Findings of the overall qualification programme (percentage over total findings)

| | **Major/ critical safety** | **Minor safety** | **Significant violation** | **Minor violation** | **All** |
|---|---|---|---|---|---|
| Integrity static analysis | 0 | 9.5 | 7.1 | 39 | 55.6 |
| Additional analysis | 2.4 | 10 | 0 | 5.9 | 18.3 |
| Field experience | 14.9 | 11.2 | 0 | 0 | 26.1 |
| Total | 17.3 | 30.7 | 7.1 | 44.9 | 100 |

Generally speaking these analyses identified different classes of faults from those found by our static analysis, and these fault classes were the ones that resulted in significant failures. This is not surprising, as static analysis will contain a significant proportion of “lurking” problems that might not cause any failure until the code is modified. Furthermore, static analysis came after these field experience faults had been found, so they had been corrected by the time static analysis took place.

### 5.2 Effects of faults on reliability and safety

While no fault is desirable, faults may be tolerable if there is a low probability of occurrence. Since the software has been subjected to at least $3x10^6$ hours of operation, the remaining faults will tend to have a low (or zero) probability of activation. There

is a theory for estimating the worst case MTTF given an estimate of residual faults (N) and operating time (T) [9]: MTTF>eT/N, where e is the exponential constant. Under the most optimistic assumptions (perfect fault reporting and correction) about 90 residual safety-related faults and $3x10^6$ hours of operation would give a worst case estimate for the MTTF for the software of around $10^5$ hours for dangerous failures. If the shutdown process has to operate for 100 hours with a probability of failure during that interval of $10^{-3}$ the MTTF should be $10^5$ hours (or around 10 years). The result could however be less if the software had been subjected to significant upgrades and extensions, rather than simple bug fixes.

Field data was also analysed to demonstrate that all modules used in the safety-related application are stable and have accumulated sufficient operating experience without modification or bug-fixes. This gives a direct indication of time to failure for the selected modules. Both analyses give some confidence that the MTTF for the most safety-critical functions will exceed the target level of $10^5$ hours.

## 6 Conclusion

This paper described the static analysis process used to assess the integrity of the source code of a COTS system (comprising 100k lines of C and 20k of assembler). This provided additional evidence of the software quality as a contribution to the overall validation programme for the COTS system. The analysis addressed the internal integrity of the code and intra-component integrity.

Integrity static analysis is feasible for industrial scale software and even for heterogeneous code, this type of analysis does not require unreasonable resources. However, the analysis process needs to be supported by tools to automate the analyses and to manage, classify and track the analysis findings.

The analysis made a major contribution to the safety justification of the industrial COTS system—finding over half the faults detected in the qualification programme for the COTS software. It also showed that most faults had little impact on safe operation—hence increasing confidence that the software was suitable for a safety related application.

All the C analysis tools had limitations, such as limits to the amount of code that could be analysed, inability to analyse some dialects of C and inability to detect some code anomalies. The use of diverse tools for the same analysis boosts confidence in the results. While it was possible to dismiss some anomalies reported by the tools automatically, a combination of automatic and manual analysis is needed to resolve most of the reported anomalies. Most of the manual analysis can be done without domain knowledge, just by intelligent code reading. However, the final resolution requires domain expertise.

We propose that static analysis has an important role to play within an approach to the assurance of COTS products. In the case where full verification of the complete system is not mandatory, static analysis can be used to justify the partitioning of the system, specialising on a small class of faults or specialising on certain behaviours. Implementing this strategy will need to take into account the pragmatic issues of scale, complexity and tool availability highlighted in this paper. The focus of the strategy should be driven by both the capabilities of the emerging technology—the static analysis landscape is developing rapidly—and by the need to focus on classes of

faults that are important in practice and that affect real-time behaviour. The selection of tools to support the analysis should be supported by common benchmarks for static analysis.

## References

[1] P G Bishop, R E Bloomfield, Tim Clement, and Sofia Guerra. "*Software Criticality Analysis of COTS/SOUP"*. In S.Anderson et al. (Eds.), SAFECOMP 2002, LNCS 2434, pp. 198-211, 2002.

[2] N. J. Ward. "The Rigorous Retrospective Static Analysis of the Sizewell 'B' Primary Protection System Software". In *Proceedings of the 12th International Conference on Computer Safety, Reliability and Security, Safecomp 93*. October 1993.

[3] S. Morton. "A Symptom of the Cure: Safer Language Subsets and Safe-Code Development". MISRA Guidelines Forum, October 2001.

[4] *Guidelines for the use of the C language in vehicle based software.* MISRA, 1998.

[5] *PolySpace Technologies*, http://www.polyspace.com/.

[6] *CodeSurfer user guide and technical reference*. Version 1.0, Grammatech, 1999.

[7] L. Hatton, *Safer C*. McGraw Hill, 1995.

[8] P. Cousot and R Cousot. "*Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by construction or approximation of fixpoints*". In Proceedings 4th ACM Symposium on Principals of Programming Languages, POPL77, ACM Press, 1977.

[9] P. G. Bishop and R. E. Bloomfield, " A Conservative Theory for Long-Term Reliability Growth Prediction" *IEEE Trans. Reliability*, vol. 45, No 4, Dec 1996.

*All necessary clearances for the publication of this paper have been obtained. If accepted, the author will prepare the final manuscript in time for inclusion in the conference proceedings and will present the paper at the conference.*