Computers & Security (2006) **25**, 221–228

ELSEVIER

Computers & Security

www.elsevier.com/locate/cose

An empirical examination of the reverse engineering process for binary files

Iain Sutherland [a,*], **George E. Kalb** [b], **Andrew Blyth** [a], **Gaius Mulley** [a]

[a] *School of Computing, University of Glamoran, Treforest, Wales, UK*
[b] *The Johns Hopkins University, Information Security Institute Baltimore, Maryland, USA*

Received 18 November 2004; accepted 4 November 2005

KEYWORDS
Reverse engineering;
Software protection;
Process metrics;
Binary code;
Complexity metrics

Abstract Reverse engineering of binary code file has become increasingly easier to perform. The binary reverse engineering and subsequent software exploitation activities represent a significant threat to the intellectual property content of commercially supplied software products. Protection technologies integrated within the software products offer a viable solution towards deterring the software exploitation threat. However, the absence of metrics, measures, and models to characterize the software exploitation process prevents execution of quantitative assessments to define the extent of protection technology suitable for application to a particular software product. This paper examines a framework for collecting reverse engineering measurements, the execution of a reverse engineering experiment, and the analysis of the findings to determine the primary factors that affect the software exploitation process. The results of this research form a foundation for the specification of metrics, gathering of additional measurements, and development of predictive models to characterize the software exploitation process.

Introduction

Deployed software products are known to be susceptible to software exploitation through reverse engineering of the binary code (executable) files. Numerous accounts of commercial companies reverse engineering their competitor's product, for purposes of gaining competitive advantages, have been published (Bull et al., 1995; Chen, 1995; Tabernero, 2002). Global movement towards the use of industrial standards, commercially supplied hardware computing environments, and common operating environments achieves software engineering goals of interoperability, portability, and reusability. This same global movement results in a reduced cost of entry for clandestine software exploiters to successfully reverse engineer a binary code file. A software exploiter, with rudimentary skills, possesses a threat to recently deployed commercial software product because (1) machine-code instruction set and executable file

* Corresponding author.
E-mail address: isutherl@glam.ac.uk (I. Sutherland).

0167-4048/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2005.11.002

formats (Tilley, 2000) are routinely published, (2) hex editors, dissemblers, software in-circuit emulators tools are readily available via Internet sources, and (3) similar attack scenarios involving reverse engineering of binary code files are readily accessible through numerous hacking websites. There are also legitimate reasons for reverse engineering code in such cases as legacy systems (Muller et al., 2000; Cifuentes and Fitzgerald, 2000) and so there is a body of published academic material (Weide et al., 1995; Interrante and Basrawala, 1988; Demeyer et al., 1999; Wills and Cross, 1996; Gannod et al., 1988) to which a software exploiter could refer although the main focus of this effort is at source code level (Muller et al., 2000).

The commercial software product developer is forced to employ various protection technologies to protect both the intellectual property content and the software development investment represented by the software asset to be released into the marketplace. The commercial software product developer must determine the appropriate protection technologies that are both affordable and supply adequate protection against the reverse engineering threat for a desired period of performance.

The absence of predictive models that characterize the binary reverse engineering software exploitation process precludes an objective and quantitative assessment of the time since first release of the software asset to when software exploitation is expected to successfully extract useful information content. Similar to parametric software development estimation models (e.g., COCOMO), the size and complexity of the binary code file to be reverse engineered are considered to be a prime contributing factor to the time and effort required to execute the reverse engineering activity. Additionally, the skill level of the software exploiter is also considered to be a primary contributing factor. This paper describes the execution of an experiment to derive empirical data that will validate a set of proposed attributes that are believed to be the primary factors affecting the binary reverse engineering process.

Background

An insider is assumed to have access to developmental information resources pertaining to the commercial software product including the product source code. An outsider does not have access to this information and must resort to analysis of available software product resources. Such available software product resources may be little more than the binary code file as released from the original developer. The outsider is forced to execute a binary reverse engineering activity beginning with the binary code file and concluding when some desired end goal has been achieved.

The entry criterion is defined as the time when the outsider first obtains a copy of the binary code file so as to commence the reverse engineering process. The commercial software product vendor must assume that this entry criterion coincides with the first market release of the product.

The exit criterion is determined by the time when the outsider has satisfied a particular end goal for the software exploitation process. Unlike software development activities where the singular end goal is to deliver a reasonably well-tested software product to an end user given the available funding and schedule resources, binary reverse engineering activities may have multiple software exploitation end goals (Kalb). The first software exploitation end goal is defined as obtaining sufficient information regarding the software product's operational function, performance, capabilities, and limitation. Satisfying this first software exploitation end goal enables the software exploiter to transfer the information gathered to other software products that are either in development or are already deployed. The second software exploitation end goal builds upon the first and is defined as enabling minor modifications to alter/enhance the deployed software product. Satisfying this second software exploitation end goal enables (1) circumvention of existing performance limiters and protection technologies to enhance the operational performance of the deployed software product, and/or (2) insertion of malicious code artefacts to corrupt the execution of the deployed software product. The third software exploitation end goal builds upon the previous two and is defined as enabling major modifications to enhance the operational performance of the deployed software product. Satisfying this third software exploitation end goal enables a significant alteration of the deployed software product's functional and operational performance characteristics.

Regardless of the particular software exploitation end goal to be obtained, the software exploitation process must be defined to base a series of experiments that will enable the capturing of measurement data. This software exploitation process commences when the exploiter acquires the binary code file that represents the subject for the reverse engineering activity. For network-centric computing, this acquisition step is rather expediently performed and may be no more effort than locating the particular executable or load file that will be the subject of subsequent reverse engineering activities. For commercial software

products, this acquisition step encompasses the purchase and installation of the product followed by the selection of a particular executable or load file for subsequent reverse engineering activities. Embedded computer systems may require greater effort during the acquisition step since the binary code assets must be extracted from internal memory devices using various attack scenarios (e.g., in-circuit emulators, bus monitors or invasive memory read-out attacks).

The next step in the software exploitation process is a static analysis of the binary code file to derive information to support subsequent reverse engineering activities (Kalb). Using a hex editing tool the software exploiter can identify useful text strings that may encompass library function names, symbol table entries, debug messages, error messages, I/O messages, and/or residual text inserted by the compilation environment (e.g., compiler version number, data and time stamps, etc.). The software exploiter can analyze the file header information used during the loading process to verify the binary file format employed (e.g., COFF, ELF, PE, etc.). Knowledge of the binary file format enables correct navigation through the contents of the binary code file along with identification of the major structural segments contained within the binary code file such as the instruction segment. Static analysis may include using a disassembler to produce human readable assembly code for sections of the instruction segment that may be analyzed to determine functional attributes.

The next step in the software exploitation process is a dynamic analysis of the binary code file to evaluate the operational characteristics of the software product (Kalb). Execution of the binary code file either on the actual target processor or within an emulation environment enables the observation of the execution behaviour of the software product. Test case inputs can be supplied to stimulate functionality within the software product wherein the execution behaviour may be observed by the software exploiter. The information gathered through static and dynamic analyses of the binary code file is sufficient for the software exploiter to achieve the first software exploitation end goal.

Achieving the second or third software exploitation end goal requires modification of the software product. The software exploiter uses the information gathered through static and dynamic analyses of the software product to determine the nature and location of the desired change/enhancement to be applied to the software product. The actual application of the change/enhancement takes the form of a software patch of the existing binary code file to alter the execution of the software product. The extent of the modifications determines whether it is the second or the third software exploitation end goal that is to be achieved.

Anticipating the software exploitation of the deployed software product, the commercial software product developer can perform a vulnerability assessment culminating with the selection of appropriate tamper resistance technologies to be integrated into the end product. The vulnerability assessment concludes with an estimate of the time since first deployment of the software product when it is anticipated that software exploiters would have achieved one of the software exploitation end goals. Based upon the estimate of software exploitation timeline, the commercial software product developer may elect to employ software tamper resistance technology. The application of software tamper resistance technology extends the software exploitation timeline by increasing the difficulty relating to reverse engineering of the binary code file contents.

Experiments have been used in the past to perform both tool assessments and user studies (Cifuentes and Fitzgerald, 2000; Gleason, 1992; Storey et al., 1996). The experiment described in this paper attempts to determine the primary factors that affect the software reverse engineering process. These primary factors once defined and characterized could be used to quantitatively estimate the software exploitation timeline diminishing the subjectivity that currently dominates the estimation process.

Assertions

Prior to executing the reverse engineering experiment, a set of assertions were identified to be validated once experimental results had been obtained. The first assertion was that a statistical model could illustrate the relationship between education and technical ability of the software exploiter and their ability to successfully reverse engineer a software product. The second assertion was that the complexity of the binary code file is related to the complexity of the human readable source code. The reverse engineering experiment uses the Halstead and McCabe software complexity metrics to explore this relationship.

Experiment

The reverse engineering experiment requires a set of test subjects to perform a sequence of tasks relating to the reverse engineering of a set of

binary code files. The test subject's progress and success during each task are monitored using a variety of techniques to enable a series of deductions to be made concerning the effort required to reverse engineer a binary code file of known size and complexity. To expediently execute the reverse engineering experiment, each task was allotted a specific amount of time. The progress of each test subject towards achieving the task objective is then assessed. This approach avoids the potentially open ended approach of allowing each test subject to perform the task to a completion criterion consuming as much time as required to complete the task.

The set of test subjects included 10 student volunteers attending the University of Glamorgan. This included six undergraduates (three second-year students and three third-year students), three masters students, and one post-masters student providing diversity in the education/technical skills suitable for experimental requirements. Prior to the commencement of the experiment the test subjects were informed that the nature of the experiment related to reverse engineering of executable programs that contained simple algorithms. The test subjects were provided a reading list and a copy of the platform used (Redhat 7.2 GNU/Linux) along with documentation.

The reverse engineering experiment is partitioned into three stages that include an initial assessment of the test subject's knowledge/skill base, execution of the reverse engineering tasks on a set of test objects, and a post-experiment assessment to obtain feedback on the experiment. A set of six test object programs were developed that included (1) Hello World, (2) Date, (3) Bubble Sort, (4) Prime Number, (5) LIBC, and (6) GCD (Table 1). The test object programs were purposely selected to be easily recognizable algorithms, approximately same size to afford reasonable reverse engineering progress given a restrictive amount of time, and absence of proprietary software elements to avoid legal infringements associated with reverse engineering of binary code files. A subset of the six test object programs were compiled with the debug option enabled (Program Set A) while another subset of the six test objects were compiled with the debug option disabled (Program Set B). This approach provides the test subjects the opportunity to reverse engineer the same test object thereby enabling the assessment of the value that debug information retained in the binary code file adds to the reverse engineering process.

The initial assessment of the test subject's knowledge/skill base requires each test subject to complete a questionnaire. The questionnaire inquired as to the number of years of experience the test subject possessed regarding UNIX and the C programming language. The majority of test subjects had at least one year's experience with UNIX and the C programming language. The questionnaire also included a series of multiple choice questions. The multiple choice questions focused on UNIX commands relating to reverse engineering to provide an assessment of the test subject's level of experience/capability.

The execution of the reverse engineering experiment required each test subject to perform a static, dynamic, and modification task on each of the test object programs within a constrained time limit. Test object filenames were selected so as not to reveal the function of the binary. Each test subject was supplied with a tutorial worksheet that provided general guidance during each specific task. For example, the static task tutorial worksheet requested each test subject to determine the size of the binary, determine the creation time of the binary, speculate as to the type of information contained in the file, identify all strings and any constants present in the executable, and generate the assembly language for the program. The dynamic task tutorial worksheet requested each test subject to determine if any input is required by the binary, describe the output produced by the binary, identify any command line arguments required by the binary, and describe the function/purpose of the binary. The modify task tutorial worksheet requested each test subject to perform a specific modification to the test object program that requires the development and insertion of a software patch to the binary code file. For example, the test subjects were requested to modify the Hello World binary so that upon execution the program would output ''World Hello'' or to modify the Bubble Sort binary so that upon execution the program sorts in descending rather than ascending order. During the time allotted for each task the test subjects were required to perform the work requested and record their findings on the tutorial worksheets provided for that task. Upon expiration of the allotted time the tutorial worksheets were collected and replaced with the next tutorial worksheet in the experiment.

Test subjects were provided with Program Set A during the morning session of the reverse engineering experiment. Experiment developers were present to observe the execution of the experiment and to observe any interactions between test subjects. Test subjects were allowed to interact during the lunchtime break since it was decided

Table 1 Reverse engineering experiment framework

Session	Event	Test object	Program function	Task	Duration (min)	Total duration (min)
Morning session	Initial assessment					
	Program Set A (debug option enabled)	1	Hello World	Static	15	35
				Dynamic	10	
				Modify	10	
		2	Date	Static	10	30
				Dynamic	10	
				Modify	10	
		3	Bubble Sort	Static	15	45
				Dynamic	15	
				Modify	15	
		4	Prime Number	Static	15	45
				Dynamic	15	
				Modify	15	
Lunch						
Afternoon session	Program Set B (debug option disabled)	5	Hello World	Static	10	30
				Dynamic	10	
				Modify	10	
		6	Date	Static	10	30
				Dynamic	10	
				Modify	10	
		7	GCD	Static	15	45
				Dynamic	15	
				Modify	15	
		8	LIBC	Static	15	45
				Dynamic	15	
				Modify	15	
	Exit questionnaire					

that some limited collaboration on experimental results would emulate real world conditions present during actual software exploitation activities. Test subjects were provided with Program Set B during the afternoon session of the reverse engineering experiment. The experiment developers were again present to observe any interactions between test subjects.

To further observe the test subject activities during the execution of the reverse engineering experiment, the test developers employed an automated screen capture tool (Camtasia) to provide a permanent record of activities. The reverse engineering experiment platform involved an Intel-based computer executing Linux Redhat 7.2 within a VMWare virtual environment hosted on Windows NT4. This enabled the complete experimental environment to be retained for future analysis and included Bash histories of command line instructions, and all temporary and history files arising from Internet accesses. The screen captures, Bash histories, temporary and history files coupled with the initial questionnaire and tutorial worksheets, provide a detailed accounting of the test subject activities.

At the completion of Program Set B the test subjects were provided an exit questionnaire to enable post-experiment assessment. The exit questionnaire assessed the amount of materials supplied on the reading list that were actually used by test subjects during the experiment along with general comments pertaining to the various stages of the reverse engineering experiment.

Results

The measurements collected during the reverse engineering experiment are analyzed to validate the two assertions defined in the beginning of this paper (section *Assertions*).

Education/technical ability

The first assertion to be validated by the experimental results concerned whether the use of a statistical model could illustrate the relationship between education and technical ability of the software exploiter and their ability to successfully reverse engineer a software product. This assertion

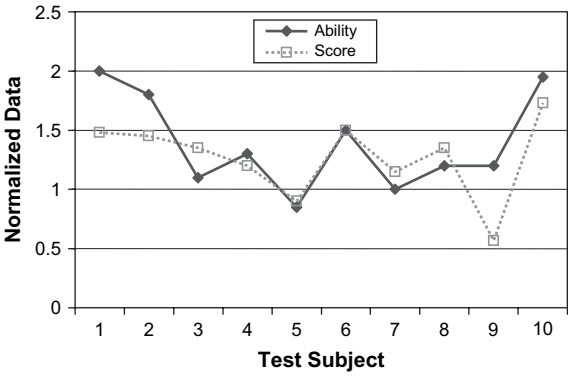

Figure 1 Grading scheme used to normalize responses.

is validated through analysis of the initial questionnaire and tutorial worksheet responses. The education/technical ability (Fig. 1, 'ability') is derived from the initial questionnaire responses for each test subject and is normalized to values between 0 and 3 (Table 2) based on their experience with operating systems, platforms, and the range of commands used during the reverse engineering experiment. The ability to successfully reverse engineer a software product (Fig. 1, 'score') is derived from the tutorial worksheet responses for each test subject and is normalized by applying a consistent grading scheme per question response (Table 2) then averaging over all of the responses (3 tasks $\times$ 8 test objects) for that particular test subject. The education/technical ability and the ability to successfully reverse engineer a software product values are plotted against the test subject's identification number. Although the two graphs do not coincide one-for-one, a correlation coefficient of 0.7236642 was computed illustrating a statistically significant relationship between the educational/technical ability of the software exploiter and their ability to successfully reverse engineer the binary code file of a software product. This result provides validation evidence for the first experiment assertion.

Table 2 Grading scheme used to normalize responses

Grade	Description
0	The test subject has failed to answer the questions, or the answer is completely incorrect.
1	The test subject has failed to demonstrate an adequate understanding of the problem. There is some factual information presented, but there may be significant errors. The answer provided by the test subject lacks sustentative matter.
2	Demonstrates an adequate understanding of the major issues and the complexity of the issues involved. The answer provided by the test subject is correct, but it may contain minor errors.
3	Demonstrates an excellent understanding of the problem and the complexity of the issues involved.

Complexity/size metric

The second assertion to be validated by the experimental results concerned the relationship between the complexity of the binary code file to the complexity of the human readable source code. This assertion is validated through correlation of the tutorial worksheet responses (regarding the reverse engineering of the eight test objects) versus the application of Halstead and McCabe metrics on the human readable source code (six software programs that when compiled produced the eight test objects). The tutorial worksheet responses for the static, dynamic, and modification tasks were normalized using the grading scheme (Table 2) then averaged to produce the mean grade per test object (3 tasks $\times$ 10 test subjects). The Halstead and McCabe metrics were computed using the source code for each of the test objects. The mean grade per test object is correlated with each of the individual metric items to determine the extent of any dependencies (Tables 3 and 4).

The statistical analysis reveals that there are no significant positive correlations between the source code metrics and the ability of the software exploiter to successfully reverse engineer a software product. The lack of correlation illustrates that source code artefacts that contribute to size and complexity metrics do not impact the reverse engineering process applied to binary code files. For example, the amount of branching (decision points) within a source code file is the basis of the McCabe cyclomatic complexity metric and has significant bearing on unit-level testing of the software module. Comparatively, branching instructions (jump instructions) within a binary code file are easily disassembled and understood by the software exploiter.

Conclusion

The reverse engineering experiment as defined within this paper represents a framework for the experimental collection of measurement data in

Table 3 Source code metrics debug enabled

Source program	Hello World	Date	Bubble Sort	Prime Number	Correlation
Test object	1	2	3	4	
Mean grade per test object	1.483	1.300	0.786	0.867	
Metric					
Lines of code	6	10	9	21	−0.5802
Software length[a]	7	27	14	33	−0.3958
Software vocabulary[a]	6	14	11	15	−0.5560
Software volume[a]	18	103	48	130	−0.4006
Software level[a]	0.667	0.167	2.5	0.094	−0.4833
Software difficulty[a]	1.499	5.988	5.988	10.638	−0.7454
Effort[a]	27	618	120	1435	−0.3972
Intelligence[a]	12	17	19	15	−0.6744
Software time[a]	0.001	0.001	0.001	0.001	0
Language level[a]	8	2.86	7.68	1.83	0.1909
Cyclomatic complexity	1	1	1	3	−0.4802

[a] Halstead metrics.

a consistent and repeatable fashion. The 10 test subjects participating in the actual reverse engineering experiment, although representing a relatively small data set, provide the basis of a preliminary assessment as to the primary factors that affect the software reverse engineering process. The reverse engineering experiment provides quantitative evidence that there is a relationship between the education/technical ability of the software exploiter and their ability to successfully reverse engineer a software product. This evidence provides the foundation for modelling of this relationship using existing predictive models. Development and maturation of a reverse engineering model that characterizes the software exploitation process will enable commercial software product developers to quantitatively predict the time following product deployment when it is anticipated that a software exploiter would have achieved a given exploitation end goal.

The reverse engineering experiment also provides quantitative evidence that industry accepted source code size and complexity metrics are not suitable for characterizing the size and complexity of binary code files pursuant to estimating the time required to perform software exploitation activities. Literary research conducted at the commencement of this project did not identify binary size and complexity metrics that could have been used instead of the source code size and

Table 4 Source code metrics debug disabled

Source program	Hello World	Date	GCD	LIBC	Correlation
Test object	5	6	7	8	
Mean grade per test object	1.350	1.558	1.700	1.008	
Metric					
Lines of code	6	10	49	665	−0.3821
Software length[a]	7	27	40	59	−0.3922
Software vocabulary[a]	6	14	20	21	−0.0904
Software volume[a]	18	103	178	275	−0.4189
Software level[a]	0.667	0.167	0.131	0.134	−0.1045
Software difficulty[a]	1.499	5.988	7.633	7.462	0.0567
Effort[a]	27	618	2346	5035	−0.5952
Intelligence[a]	12	17	17	19	−0.1935
Software time[a]	0.001	0.001	0.2	0.4	−0.5755
Language level[a]	8	2.86	2.43	2.3	−0.0743
Cyclomatic complexity	1	1	3	11	−0.7844

[a] Halstead metrics.

complexity metrics. Size and complexity metrics that directly characterize the binary code files must be defined. Such size and complexity metrics are required to support the development of a software exploitation predictive models a follow-on research project has been proposed to define these metrics and then use the existing reverse engineering experiment framework to gather measurements to corroborate the defined metrics.

Acknowledgment

The researchers wish to thank the sponsor of this project, who requested to remain anonymous, for the generous funding of this project and for providing funding for the follow-on research project.

References

Bull TM, Younger EJ, Bennett KH, Luo Z. Bylands: reverse engineering safety-critical systems. In: Proceedings of the international conference on software maintenance. IEEE Computer Society Press; October 17–20, 1995. p. 358–66.

Chen Y. Reverse engineering. In: Practical reusable UNIX software. John Wiley & Sons; 1995.

Cifuentes C, Fitzgerald A. The legal status of reverse engineering of computer software. Annals of Software Engineering 2000;vol. 9. Baltzer Science Publishers.

Demeyer S, Ducasse S, Lanza M. A hybrid reverse engineering approach combining metrics and program visualisation. Published in. In: The proceedings of working conference on reverse engineering 1999. Washington DC, USA: IEEE Computer Society; 1999.

Gannod G, Chen Y, Cheng B. An automated approach for supporting software reuse via reverse engineering. In: Thirteenth international conference on automated software engineering. IEEE Computer Society Press; 1988. p. 94–104.

Gleason JA. A reverse engineering tool for a computer aided software engineering (CASE) system. Technical Report. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science; 1992.

Interrante MF, Basrawala Z. Reverse engineering annotated bibliography technical report. Software Engineering Research Centre; 26 January 1988. Number SERC-TR-12-F.

Kalb G. Embedded computer systems – vulnerabilities, intrusions and protection mechanisms courseware. The Johns Hopkins University Information Security Institute.

Muller H, Jahnke J, Smith D, Storey Margaret-Anne, Tilley S, Wong K. Reverse engineering: a roadmap. In: ACM 2000; 2000.

Storey MAD, Wong K, Fong P, Hooper D, Hopkins K, Muller HA. On designing an experiment to evaluate a reverse engineering tool. Published in. In: The proceedings of working conference on reverse engineering 1996. Washington DC, USA: IEEE Computer Society; 1996.

Tabernero M. Embedded system vulnerabilities & The IEEE 1149.1 JTAG standard, <http://www.cs.jhu.edu/~kalb/Kalb_JTAG_page.htm>; February 2002.

Tilley SR. The canonical activities of reverse engineering. Annals of Software Engineering 2000;vol. 9. Baltzer Science Publishers.

Weide BW, Heym WD, Hollingsworth JE. Reverse engineering of legacy code expose. In: Proceedings: 17th international conference on software engineering. IEEE Computer Society Press/ACM Press; 1995. p. 327–31.

Wills LM, Cross II JH. Recent trends and open issues in reverse engineering. Automated Software Engineering: An International Journal July 1996;vol. 3(1/2):165–72. Kluwer Academic Publishers.

Further reading

<http://www.wotsit.org>; July 1996.

Iain Sutherland is a lecturer in the Information Security Research Group at the School of Computing, University of Glamorgan, UK. His main research interests are Information Security and Computer Forensics. Dr. Sutherland received his Ph.D. from Cardiff University.

George E. Kalb is an Instructor and Institute Fellow at the Johns Hopkins University Information Security Institute, US. His research interests are in the domains of binary reverse engineering and tamper resistance technologies. He has a B.A. in Physics and Chemistry from University of Maryland and an M.S. in Computer Science from Johns Hopkins University.

Andrew Blyth is currently the Head of the Information Security Research Group at the School of Computing, University of Glamorgan, UK. His research interests include network and operating systems security, and reverse engineering. Dr. Blyth received his Ph.D. from Newcastle University.

Gaius Mulley is a senior lecturer at the University of Glamorgan. He is the author of GNU Modula-2 and the groff html device driver grohtml. His research interests also include performance of micro-kernels and compiler design. Dr. Mulley received his Ph.D. and B.Sc.(Hons) from the University of Reading.