Code Analysis: Past and Present

Daniela da Cruz[1], **Pedro Rangel Henriques**[2] **and Jorge Sousa Pinto**[3]

[1] danieladacruz@di.uminho.pt, http://www.di.uminho.pt/~danieladacruz
[2] prh@di.uminho.pt, http://www.di.uminho.pt/~prh
[3] jsp@di.uminho.pt, http://www.di.uminho.pt/~jsp
Department of Computer Science
University of Minho, Braga, Portugal

Abstract: The integration of Software components within complex industrial applications with severe security standards, requires strict quality assessment of each integrated component. That is, requires a guarantee that each component is compliant with the software development *good practices* and all the standards in use. If full certification is easy to obtain for proprietary modules, it is particularly hard to achieve when dealing with Open-Source Software pieces, demanding for rigorous methods and techniques to implement their certification process.

In this context, code analysis plays an important role as the basis for the automatization of quality assessment of open source software projects – code analysis provides the techniques and tools to implement the necessary validation process. Although source code is still the most explored (the main support for analysis), nowadays this assessment process should be able to deal with code at different compilation levels.

Due to its relevance for the *open source software certification task*, this paper reviews code analysis area (stages of the analyzing process, traditional approaches, and future trends), aiming at identifying what is available, and what deserves further research.

Keywords: Code Analysis, Data Extraction, Information Representation

1 Introduction

The increasing amount of software developed in the last few years have produced a growing demand for programmers and programmer productivity to maintain it working along the years. During maintenance, the most reliable and accurate description of the behavior of a software system is its source code. Even nowadays, when modern software projects start with the construction of models (e.g. using the UML) that can be "compiled" to traditional source code, source code is still considered "the truth" and "the system" (because the generated code is incomplete and requires that the programmers complete it by hand).

So, *Source Code Analysis*—according to David Binkley, in [Bin07], *the process of extracting information about a program from its source code or artifacts generated from the source code using automatic tools*—is crucial to support maintenance. However, given the complexity of modern software, the manual analysis of code (source code, intermediate, or machine code), is costly and ineffective. A more viable solution is to resort to tool support. Such tools provide

information to programmers that can be used to coordinate their efforts and improve their overall productivity. The information provided is composed by data items such as: *Identifiers Table*, *Abstract Syntax Tree* (AST) or *Decorated Abstract Syntax Tree* (AST), *Control Flow Graph* (CFG), *Value Dependence Graph* (VDG), *Call Graph*, *Module Dependence Graph* (MDG), *Trace Flow Graph* (TFG), *Static Single Assignment* (SSA), etc. Those tools must be able to cope with the two complementary approaches to code analysis: static and dynamic. In both of them, the extracted information must be coherent with the language semantics, in order to help a programmer gain insight of the source code's meaning.

In this context, we are not concerned with software maintenance; instead, our motivation is the open-source software certification. However we strongly believe that code analysis methods and the knowledge extractable with them (distributed by components like those listed above) can also support the OSS verification process.

The remainder of this chapter is organized as follow. In section 2 are presented the stages of a typical code analysis. In section 4 some current code analysis techniques are discussed. We close the paper with some remarks and research trends in section 5.

2 Code Analysis Anatomy

Under the umbrella of code analysis, there are many techniques used to handle static and dynamic information relevant to characterize the syntax (structures) and semantics (behavior) of a program. However this mess of algorithms and data structures can be organized in three main components needed for code analysis: a) data extraction; b) information representation; and c) knowledge exploration. These components, their purpose and possible approaches and implementations will be discussed in the rest of this section.

2.1 Data extraction

The process of retrieving data out of data sources for further data processing or data storage is named data extraction. The export of that data into an intermediate representation is a common strategy to make easier the data analysis/transformation and possibly the addition of metadata, facilitate prior to export to another stage in the data workflow.

In the context of code analysis this process is usually executed by a syntactic analyzer, or parser. It parses the code into one or more internal representations. A parser is the part of a compiler that goes through a program and cuts it into identifiable parts (commonly called *chunks*) before translation, each chunk more understandable than the whole. Basically, the parser searches for patterns of operators and operands to group the source string into smaller but meaningful chunks.

Parsing is the necessary evil of most code analyses. While not theoretically difficult, the complexities of modern programming languages, in particular those that are not LR(1) [AU72] and those incorporating some kind of preprocessing make harder code analysis significantly, as will be seen in section 4.1. Parsers are supported by lexical analyzers that convert character sequences into words (the language terminal symbols) and extract their real semantic value. They are complemented by semantic analyzers that evaluate the concrete meaning of the chunks.

2.2 Information representation

After extracting from the code the relevant information, there is a need to represent it in a more abstract form. This is the second component of code analysis: storage of the collected data into an internal representation (IR), such that data is kept grouped in meaningful parts and the relations among them are also stored to give sense to the whole. The main goal of this phase is to abstract a particular aspect of the program into a form more suitable for automated analysis. Essentially, an abstraction is a sound, property-preserving transformation to a smaller domain. Some internal representations are produced directly by the parser (e.g. Abstract Syntax Trees (AST), Control Flow Graphs (CFG), etc), while others require the result of prior specific analyses (e.g., dependence graphs requires prior pointer analysis).

Many internal representations raise from the compilers area. Generally, the most common internal representations use graphs (specially if they degenerate in forms such as trees) — the most widely used are the Control Flow Graph (CFG), the Call Graph. A Value Dependence Graph (VDG) is another graph variant that improves (at least for some analysis) the results obtained using the Single Static Assignment (SSA) form. VDGs represent control flow as data flow and thus simplify analyses [WCES94].

Another commonly used graph is the Dependence Graph, introduced in the context of work with parallelizing and highly optimizing compilers [FOW87], where vertices represent the statements and predicates of a program. These graphs have been used in other analyses [HRB88, HR92]. A related kind of graph, the Module Dependency Graph (MDG), used by Bunch tool, represents programs at a coarser level of granularity. Its vertices represent modules of the system and edges represent the dependencies between them [MMCG99].

Other sorts of graphs, also referred in the literature, include Dynamic Call Graphs [QH04, PV06] (is intended to record an execution of a program) and XTA graphs built in support of dynamic reachibility-based interprocedural analysis [QH04]. These techniques are required to analyze code written in languages such as Java that include dynamic class loading.

Finally, the Trace Flow Graph is used to represent concurrent programs [CCO01].

All of the variants of graphs or other internal representations presented are actually used according to the type of analysis and the desired results of that analysis. In real applications, it is common to combine different kinds of graphs or AST with Identifier Tables (or similar mapping) in such a way that enriches and structures the information extracted.

2.3 Knowledge Exploration

After organizing the data extracted into an intermediate representation that makes or transforms it into information, the third component of code analysis is aimed at knowledge inference. This process requires that the pieces of the information be stored, interconnected and also be interrelated with previous knowledge. This can be achieved using quantitative or qualitative methods. Concerning quantitative methods, resorting to program metrics is the most commonly used approach. Concerning qualitative methods, name analysis, text and data mining, and information retrieval are the most widely used. Visualization techniques are crucial for the effectiveness of this process.

3 Code Analysis Strategies

According to Binkley [Bin07], the main strategies for code analysis could be classified as follows: *static* versus *dynamic*, *sound* versus *unsound*, *flow sensitive* versus *flow insensitive*, and *context sensitive* versus *context insensitive*.

3.1 Static vs dynamic

Static analyses analyze the program to obtain information that is valid for all possible executions. Dynamic analyses instrument the program to collect information as it runs. The results of a dynamic analysis are typically valid for the run in question, but result no guarantees regarding other runs. For example, a dynamic analysis for the problem of determining the values of global variables could simply record the values as they are assigned. A static analysis might analyze the program to find all statements that potentially affect the global variables, then analyze the statements to extract information about the assigned values.

Dynamic analysis has the advantage that detailed information about a single execution is typically much easier to obtain than comparably detailed information that is valid over all executions.

Another significant advantage of dynamic tools is the precision of the information that they provide, at least for the execution under consideration. Virtually all static analyses extract properties that are only approximations of the properties that actually hold when the program runs. The trade-off, of course, is that the properties extracted from one execution may not hold in all executions; the size of dynamic data extracted could also be a handicap.

Some techniques sit in between and most analyses require their combination.

3.2 Sound vs unsound

A deductive system is sound with respect to a given semantics if it proves valid arguments only. So, a sound analysis makes correctness guarantees. Sound static analyses produce information that is guaranteed to hold on all program executions; sound dynamic analyses produce information that is guaranteed to hold for the analyzed execution alone.

Unsound analyses make no such guarantees. A sound analysis for determining the potential values of global variables might, for example, use pointer analysis to ensure that it correctly models the effect of indirect assignments that take place via pointers to global variables. An unsound analysis might simply scan the program to locate and analyze only assignments that use global variables directly, by name. Because such an analysis ignores the effects of indirect assignments, it may fail to compute all of the potential values of global variables.

It can be strange why an engineer will be interested in unsound analysis. However, in many cases, the information from an unsound analysis is correct, and even when incorrect, it may provide a useful starting point for further investigation. Unsound analyses are therefore often quite useful for those faced with the task of understanding and maintaining legacy code.

The most important advantages of unsound analyses, however, are their ease of implementation and efficiency. An unsound analysis may thus be able to analyze programs that are simply beyond the reach of the corresponding sound analysis, and may be implemented with a small fraction of the implementation time and effort required for the sound analysis. These points

justify the continuing importance of unsound analyses.

A slightly different concept from sound analysis is the notion of *safe analysis*. Qualifying a static analysis as *safe* means that the answer is precise on "one side".

3.3 Flow sensitive vs Flow insensitive

A *flow-sensitive* analysis takes the execution order of the program statements into account. It normally uses some form of iterative dataflow analysis to produce a potentially different analysis result for each program point. Flow-insensitive analyses do not take the execution order of the program statements into account, and is therefore incapable of extracting any property that depends on this order. They often use some form of type-based or constraint based analysis to produce a single analysis result that is valid for the entire program.

In contrast, a *flow-insensitive* analysis treats the statements of a program as an unordered collection and must produce conservative results that are safe for any order. In the above example, a flow-insensitive analysis must include in its results the fact that q might point to a or b. This reduction in precision comes with a reduction in computational complexity.

3.4 Context sensitive vs Context insensitive

Many programming languages provide constructs such as procedures that can occur in different contexts. Roughly speaking, a *context-insensitive* analysis produces a single result that is used directly in all contexts.

A *context-sensitive* analysis produces a different result for each different analysis context. The two primary approaches are to reanalyze the construct for each new analysis context, or to analyze the construct once (typically in the absence of any information about the contexts in which it will be used) to obtain a single parameterized analysis result that can be specialized for each analysis context.

Context sensitivity is essential for analyzing modern programs in which abstractions (such as abstract datatypes and procedures) are pervasive.

4 Code Analysis Challenges

In previous section, we review the traditional strategies that are actually in use. However, the intrinsic complexity of such tasks combined with the natural evolution of programming languages and systems integration, implies the existence of various open topics. In this section we present such challenges that are being posed to code analysis.

For the sake of space, for each referred challenge we just characterize it briefly, cite the work done and sum up with future trends.

4.1 Language Issues

In the last few years, many enhancements have been made to programming languages. For instance, the introduction of concepts such as dynamic class loading and reflection in languages

such as Java and C#, respectively, has contributed to advance the state of the art of programming languages.

These concepts and also the presence of casting, pointer arithmetic, anonymous types and the like make the task of parsing a difficult task.

Modern languages increasingly require tools for high precision source code analysis to handle only partially known behavior (such as generics in Java, plug-in components, reflection, user-defined types, and dynamic class loading). These features increase flexibility at run-time and impose a more powerful dynamic analysis, but compromise static analysis.

4.2 Multi-Language Analysis

Many modern software systems are heterogeneous, i.e., they are composed of modules written in different programming and specification languages. Current software development tools, e.g., Integrated Development Environments (IDEs), cannot analyze these mixed-language systems as a whole, since they are too closely related to a particular programming language and do not process mixed-language systems across language boundaries.

So, multi-language analysis grows more and more important as systems are progressively more heterogeneous. Even a simple Java program can consist of Java-source and -bytecode components. A larger system, e.g., a web application, may join SQL, HTML, and Java codes on the server site and additional languages on the client site. For example, the Visual Studio .Net environment merges languages such as ASP, HTML, C#, J#, and Visual Basic.

The key for a multi-language analysis is a common meta-model to capture the concepts of each programming language, as proposed by [SKL06]. Within this solution, the minimal set of features that need to be implemented for each new type include parsing, syntax mappings, and semantic analysis functions. Also, might be necessary to extend the constructs in order to capture properties of a new language.

4.3 Real-Time analysis

As previously referred, static analysis is usually faster than dynamic analysis but less precise. Therefore it is often desirable to retain information from static analysis for run-time verification, or to compare the results of both techniques. It would be desirable to share the same generic algorithm by static and dynamic analysis. Although many work has been done in this area [MLL05, GSH97], there other kind of analysis that should be considered: real-time analysis.

This research problem has two distinct facets: compile-time and run-time. Self-healing code[1] and instrumented code are run-time examples. Here analysis is being done real time while the program is executing. The archetypical example of this idea is *just-in-time* compilation.

Looking forward, more such processing can be done in real-time. For instance, code coverage and memory-leak analysis might be performed, at least partially, at compile time instead of at run-time. This has the advantage of providing information about a piece of code that is current focus of the programmer.

[1] While no consensus-based definition of the term “self-healing” exists, intuitively, these systems automatically repair internal faults.

4.4 Analyzing executables

In the past years a considerable amount of research activity has developed static-analysis tools to find bugs and vulnerabilities. However, most of the effort has been on static-analysis of source code, and the issue of analyzing executables was ignored. In the security context this is particularly unfortunate because source code analysis can fail to detect certain vulnerabilities, due to the phenomenon: "What You See Is Not What You eXecute" (WYSINWYX). That is, there can be a mismatch between what a programmer intends and what is actually executed on the processor.

Many efforts have been made to improve the recovery of IRs through the analysis of executables [RBL06, Wal91, RBL06]. However, there is still a need to further develop this area to cover other aspects like dynamic languages, object-oriented programming languages and so on.

4.5 Information Retrieval

In the last years, Information Retrieval has blossomed with the growth of the Internet and the huge amount of information available in electronic form. Some applications of Information Retrieval to code analysis include automatic link extraction [ZB04], concept location [MSRM04], software and website modularization [GMMS07], reverse engineering [Mar], software reuse impact analysis [SR03, FN87], quality assessment [LFB06], and software measurement [HSS01].

These techniques can be used to estimate a language model for each "document" (e.g. a source file, a class, an error log, etc) and then a classifier can be used to score each model. Much of this work has a strong focus on program identifiers [LMFB06]. Unlike other approaches that consider non-source documents, this approach focuses exclusively on the code. It divides each source code module into two documents: one includes the comments and the other the executable source code.

To date, the application of IR has concentrated on processing the text from source and non-source (which can be just as important as source) software artifacts using only a few developed IR techniques. Given the growing importance of non-source documents, source code analyses should in time develop new IR-based algorithms specifically designed for dealing with source code.

4.6 Data Mining

Recently the mining of software-related data repositories has started. Techniques such as the analysis of large amounts of data require significant computing resources and the application of techniques such as pattern recognition [PM04], neural networks [LSL96], and decision trees [GFR06], which have advanced dramatically in recent years.

Most existing techniques have been proposed by software engineering researchers, who often reuse simple data mining techniques such as association mining and clustering. A wider selection of data mining techniques should be more widely applied that removes the requirement that existing systems fit the features provided by existing mining tools.

Data mining is also being applied to software comprehension. In [KT04], the authors propose a model and associated method to extract data from C++ source code which is subsequently to be mined, and evaluate a proposed framework for clustering such data to obtain useful knowledge. It is thus clear that there is a demand for the adaptation or development of more advanced data

mining methods.

Other future challenges in code analysis will emerge, such as real-time verification; and improved support for user interaction — rather than being asked to make a collection of similar low-level choices, tools will ask about higher level-patterns that can be used to avoid future questioning. Code analysis tools will also need to use information from edit, compile, link, and run-time and continue to include a combination of multiple views of a software system such as structure, behavior, and run-time snapshots, that is what is being proposed in this thesis.

5 Conclusion

To be feasible to study Open-Source Software properties, for instance to check its compliance with programming and security standards, it is necessary to inspect the source code in order to extract its syntactic structure, its partial content and its overall meaning. This is systematically done using a set of techniques known as *code analysis* that can be grouped into three major components: data extraction; information representation; and knowledge exploration. Static Data Extraction is usually done in three steps: lexical, syntactic, and semantic analysis. Code instrumentation based methods can also be used complementary to collect dynamic (runtime) data. Data items so far obtained are gathered in appropriate data structures—*Abstract Syntax Tree, Identifiers Table*, and *Graphs*—to form an Information Repository.

Nowadays it comes to the evidence that more powerful analysis techniques are necessary to overcome the problems raised up by the existence of open-source software systems based on components available at different compilation stages (source, intermediate, and machine levels). From this fact rose up the need of a platform (with new extraction strategies and techniques) where exploration (the third analysis stage) can be done in an uniform way.

Manual or automatic inference mechanisms are then applied to that information repository to explore it, producing new Knowledge. The complexity of those inference algorithms requires a sensitive balance between *precise* and *unprecise* approaches; while the former produces complete outcomes at an high cost, the latter produces, at a reasonable cost, incomplete results that many times are satisfactory.

Code analysis was reviewed along the paper as it provides the foundations to implement the software certification process.

Bibliography

[AU72] A. V. Aho, J. D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[Bin07] D. Binkley. Source Code Analysis: A Road Map. In *FOSE '07: 2007 Future of Software Engineering*. Pp. 104–119. IEEE Computer Society, Washington, DC, USA, 2007. doi:http://dx.doi.org/10.1109/FOSE.2007.27

[CCO01] J. M. Cobleigh, L. A. Clarke, L. J. Osterweil. FLAVER: A Finite State Verification Technique for Software Systems. Technical report, Amherst, MA, USA, 2001.

[FN87] W. B. Frakes, B. A. Nejmeh. Software reuse through information retrieval. *SIGIR Forum* 21(1-2):30–36, 1987.
doi:http://doi.acm.org/10.1145/24634.24636

[FOW87] J. Ferrante, K. J. Ottenstein, J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9(3):319–349, 1987.
doi:http://doi.acm.org/10.1145/24039.24041

[GFR06] J. Gama, R. Fernandes, R. Rocha. Decision trees for mining data streams. *Intell. Data Anal.* 10(1):23–45, 2006.

[GMMS07] U. Güntzer, R. Müller, S. Müller, R.-D. Schimkat. Retrieval for decision support resources by structured models. *Decis. Support Syst.* 43(4):1117–1132, 2007.
doi:http://dx.doi.org/10.1016/j.dss.2005.07.004

[GSH97] R. Gupta, M. L. Soffa, J. Howard. Hybrid slicing: integrating dynamic information with static analysis. *ACM Trans. Softw. Eng. Methodol.* 6(4):370–397, 1997.
doi:http://doi.acm.org/10.1145/261640.261644

[HR92] S. Horwitz, T. Reps. The use of program dependence graphs in software engineering. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*. Pp. 392–411. ACM, New York, NY, USA, 1992.
doi:http://doi.acm.org/10.1145/143062.143156

[HRB88] S. Horwitz, T. Reps, D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*. Volume 23(7), pp. 35–46. Atlanta, GA, June 1988.
http://citeseer.ist.psu.edu/horwitz90interprocedural.html

[HSS01] U. Hanani, B. Shapira, P. Shoval. Information Filtering: Overview of Issues, Research and Systems. *User Modeling and User-Adapted Interaction* 11(3):203–259, 2001.
doi:http://dx.doi.org/10.1023/A:1011196000674

[KT04] Y. Kanellopoulos, C. Tjortjis. Data Mining Source Code to Facilitate Program Comprehension: Experiments on Clustering Data Retrieved from C++ Programs. *iwpc* 00:214, 2004.
doi:http://doi.ieeecomputersociety.org/10.1109/WPC.2004.1311063

[LFB06] D. J. Lawrie, H. Feild, D. Binkley. Leveraged Quality Assessment using Information Retrieval Techniques. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*. Pp. 149–158. IEEE Computer Society, Washington, DC, USA, 2006.
doi:http://dx.doi.org/10.1109/ICPC.2006.34

[LMFB06] D. Lawrie, C. Morrell, H. Feild, D. Binkley. What's in a Name? A Study of Identifiers. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*. Pp. 3–12. IEEE Computer Society, Washington, DC, USA, 2006.
doi:http://dx.doi.org/10.1109/ICPC.2006.51

[LSL96] H. Lu, R. Setiono, H. Liu. Effective Data Mining Using Neural Networks. *IEEE Trans. on Knowl. and Data Eng.* 8(6):957–961, 1996.
doi:http://dx.doi.org/10.1109/69.553163

[Mar] A. Marcus. PhD thesis.

[MLL05] M. Martin, B. Livshits, M. S. Lam. Finding application errors and security flaws using PQL: a program query language. *SIGPLAN Not.* 40(10):365–383, 2005.
doi:http://doi.acm.org/10.1145/1103845.1094840

[MMCG99] S. Mancoridis, B. S. Mitchell, Y. Chen, E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*. P. 50. IEEE Computer Society, Washington, DC, USA, 1999.

[MSRM04] A. Marcus, A. Sergeyev, V. Rajlich, J. I. Maletic. An Information Retrieval Approach to Concept Location in Source Code. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*. Pp. 214–223. IEEE Computer Society, Washington, DC, USA, 2004.

[PM04] S. K. Pal, P. Mitra. *Pattern Recognition Algorithms for Data Mining: Scalability, Knowledge Discovery, and Soft Granular Computing*. Chapman & Hall, Ltd., London, UK, UK, 2004.

[PV06] S. Pheng, C. Verbrugge. Dynamic Data Structure Analysis for Java Programs. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*. Pp. 191–201. IEEE Computer Society, Washington, DC, USA, 2006.
doi:http://dx.doi.org/10.1109/ICPC.2006.20

[QH04] F. Qian, L. Hendren. Towards dynamic interprocedural analysis in JVMs. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*. Pp. 11–11. USENIX Association, Berkeley, CA, USA, 2004.

[RBL06] T. Reps, G. Balakrishnan, J. Lim. Intermediate-representation recovery from low-level code. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. Pp. 100–111. ACM, New York, NY, USA, 2006.
doi:http://doi.acm.org/10.1145/1111542.1111560

[SKL06] D. Strein, H. Kratz, W. Lowe. Cross-Language Program Analysis and Refactoring. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. Pp. 207–216. IEEE Computer Society, Washington, DC, USA, 2006.
doi:http://dx.doi.org/10.1109/SCAM.2006.10

[SR03] E. J. Stierna, N. C. Rowe. Applying information-retrieval methods to software reuse: a case study. *Inf. Process. Manage.* 39(1):67–74, 2003.
doi:http://dx.doi.org/10.1016/S0306-4573(02)00025-0

[Wal91] D. W. Wall. Systems for Late Code Modification. In *Code Generation*. Pp. 275–293. 1991.

[WCES94] D. Weise, R. F. Crew, M. Ernst, B. Steensgaard. Value dependence graphs: representation without taxation. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Pp. 297–310. ACM, New York, NY, USA, 1994.
doi:http://doi.acm.org/10.1145/174675.177907

[ZB04] J. Zeng, P. A. Bloniarz. From Keywords to Links: an Automatic Approach. In *ITCC '04: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2*. P. 283. IEEE Computer Society, Washington, DC, USA, 2004.