Volume 4, Issue 6, June 2014 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Empirical Evaluation of Mutation Testing for Improving the Security and Reliability of Web- Application

G.Pardha Sagar*, Prof.N.D.Kale
Department of Computer
Engineering&Pun university,
India

Abstract*— In today's world as the use of web-application is increasing the need of security in web application is also increasing. Securing web-application is very important as it has critical data and wide usage .The success of any web-application depends upon the security provided in it. We use mutation testing and mutation analysis to generate empirical analysis of various parameters. These empirical results will in turn help us to recognize vulnerabilities, software components are to be addressed and tested carefully. The empirical analysis results show us all the necessary information about the web –application security parameters and hence we can test and secure the web application using these results.

***Keywords*— Mutation,Security,Web-Application,Vulnerability.**

I. Introduction

Mutation testing was initially proposed in the 1970s as it intends to guarantee vigo in test case improvement. By making syntactically right substitutions inside the software under test (SUT) and rehashing the test execution stage against the adjusted code, an evaluation could be made of test quality contingent upon if the definitive test cases could locate the code adjustment. Mutation Testing is a shortcoming based testing procedure which furnishes a testing rule called the "mutation ampleness score" or “mutation adequacy score”. The mutation ampleness score could be utilized to measure the viability of a test set as far as its capability to catch faults .The general guideline underlying Mutation Testing work is that the shortcomings utilized by Mutation Testing speak to the errors that programmers regularly make. In mutation investigation, from a program P, a set of flawed programs P’ called mutants, is produced by a couple of single syntactic changes to the definitive program P. Mutation testing (or Mutation investigation or Program mutation) is utilized to plan new programming tests and assess the nature of existing programming tests[1]. Mutation testing includes changing a program's source code or byte code in little ways. Each transformed variant is known as a mutant and tests discover and reject mutants by making the conduct of the definitive form vary from the mutant. This is called killing the mutant. Test suites are measured by the rate of mutants that they kill. New tests might be intended to kill extra mutants. Mutants are dependent upon generally characterized mutation operators that either impersonates common programming slips, (for example utilizing the wrong operator or variable name) or energy the production of significant tests, (for example driving every outflow to zero). The object is to help the analyzer advance adequate tests or spot shortcomings in the test information utilized for the program or as a part of areas of the code that are from time to time or never entered throughout execution [2]. Mutation Testing closes with a sufficiency score, known as the Mutation Score, which demonstrates the nature of the data test set. The Mutation Score (MS) is the proportion of the amount of slaughtered mutants over the aggregate number of non-equal mutants. The objective of mutation dissection is to raise the mutation score to 1, demonstrating the test set T is sufficient to recognize all the flaws indicated by the mutants [1]. Mutation Testing carries an entire new level of blunder discovery to the programming engineer. This compelling strategy has the ability to uncover ambiguities in code long ago thought difficult to catch automatically .errors that were once baffling and tedious to discover and fix can now be discovered automatically .the client additionally profits from Mutation Testing, as the project he accepts is less surrey and more solid.[3]. Need of security testing in web applications .Web provision clients and Web requisition vulnerabilities are expanding. This will inescapably uncover more Web provision clients to noxious ambushes. Security testing is a standout amongst the most vital programming security hones, which is utilized to moderate vulnerabilities in programming. Security testing of Web provisions is getting entangled, and there is still need for security testing procedures. This shows that security testing philosophies for Web requisitions needs consideration .With the pervasive of online requisitions that include the utilization of private information over the Internet, provision level security has turned into a basic concern [4][5].

As a major implies for guaranteeing programming security, different security testing methods have been produced for discovering programming vulnerabilities previously. Case in point, we have improved methods for immediately producing security tests from threat models, spoke to by threat trees, threat nets, also UML sequence diagram. In any case, benchmarks are in incredible requests for exactly measuring the helplessness recognition proficiencies of these methods. Such benchmarks may as well meet the accompanying necessities. First and foremost, they may as well be true requisitions with rich business rationale, not toy cases. This is discriminating to the assessment and correlation of commonsense utility (e.g., adaptability) of security testing methods. Second, they might be utilized to assess security testing methods for different vulnerabilities, not only a specific sort of vulnerabilities (e.g., infusion or XSS) [6]. This paper is motivated to develop such a benchmark based on Sugar CRM a fully-fledged customer relationship management web application. Sugar CRM is open source software with more than 1 million downloads, translation in multiple languages and more than 30000 community users. As it is open source and sufficiently large it provides which provides an excellent platform for mutation analysis as per our requirements. Due to the availability of source code we can create enough security mutants and analyze in a systematic way, by systematic here we means coverage of all functionalities with security concerns. Also the OWASP's ten most critical web applications security concerns should be covered as they are the main causes of vulnerabilities. We have created many security mutants of sugar CRM and applied them to empirically evaluate two automated security testing techniques. We are using penetrative testing as it is application specific and code review as it is more reliable. Mutation Testing has been progressively generally examined since it was initially proposed in the 1970s. There has been much look into deal with the different sorts of procedures trying to turn Mutation Testing into a reasonable testing methodology. There is an expansive assemblage of writing on mutation testing. The existing examination concentrates on mutant creation through syntactic progressions. By the by, there are two assemblies of take a shot at transformation for security testing [7]. One uses mutation for testing XSS and SQL infusion strike against web applications and the other for testing access control strategies.

II. RELATED WORK

Mutation Testing has been progressively generally examined since it was initially proposed in the 1970s. There has been much look into deal with the different sorts of procedures trying to turn Mutation Testing into a reasonable testing methodology. There is an expansive assemblage of writing on mutation testing. The existing examination concentrates on mutant creation through syntactic progressions. By the by, there are two assemblies of take a shot at transformation for security testing [7]. One uses mutation for testing XSS and SQL infusion strike against web applications and the other for testing access control strategies. In this paper, on the other hand, mutation examination includes different vulnerabilities connected with the functionalities of an application. Our methodology is distinctive from conventional transformation testing, which has kept tabs on mutant creation through syntactic progressions, for example, supplanting &&with ||. Syntactic progressions may not bring about significant vulnerabilities in security-escalated programming. Shahriar and Zulkernine have advanced the MUTEC and MUSIC apparatuses for mutation testing of web applications. These apparatuses have concentrated on a specific kind of vulnerabilities. Fonseca et al. Survey the viability of web filtering devices by seeding XSS and SQL infusion blames in web applications [8]. Webgoat is a deliberately shaky web application for exhibiting different security vulnerabilities, yet is not suited for benchmarking security testing methods. For mutation testing of access control arrangements, there are two principle frameworks used to compose access control approaches. One utilizes Orbac (Organization-Based Access Control) or RBAC (Role-Based Access Control) as a demonstrating plan to uniquely plan a security arrangement without the execution. Alternate utilization XACML (extensible Access Control Markup Language) to make the real execution of a security arrangement. Some research has utilized both to actualize a model. The above work concentrates on security testing of access control strategy, as opposed to general security testing of programming. Despite the fact that right to gain entrance control is a vital part of security (especially validation and approval), security concerns go outside access ability to control, for example, respectability and accessibility. In strategy testing, test inputs are appeals and test yields are answers. The shortcomings for the mutants are seeded in the strategy. By passing the appeal through the arrangement, the answers will demonstrate if the test ought to be executed by contrasting them with the first ever.Testing for security is hard in that it is difficult to test a genuine system against all unintended conducts or invalid inputs. Specifically, security testing requirements to test the" vicinity of a canny enemy intent on breaking the framework".To meet this need, our later research has exhibited that threat models can give a premise for security testing since they portray security dangers from the outlook of how the foe might ambush or adventure a framework. We propose an approach in our paper for security testing using threat model represented as threat trees. Threat trees or attack trees are a widely used representation for threat models for secure software development [9][10]. We have presented a tool for security testing covering all major security related functionalities keeping STRIDE and OWASP's top ten critical security threats for web applications. This paper aims at empirically evaluating the various approaches for security testing of web application through mutation analysis and creating a benchmark for security testing of web applications [10].

III. EXISTING TECHNIQUES

3.1 :The Open Web-Application Security Project (OWASP) testing framework

The Open Web Application Security Project (OWASP) is an open, not-for profit, neighbourhood committed to empowering associations to improve, buy, and furthermore uphold applications that could be trusted. The greater part of the OWASP instruments, reports, gatherings, and parts are free and open to anybody intrigued by making strides application security .One of the real commitments by OWASP is their security testing system for web applications. The schema is not produced for a particular SDLC, anyway is noticeably an exhaustive nonexclusive improvement model that holds the important exercises required for orderly security testing of Web applications. In view of the system's all inclusive statement, associations can pick and join those exercises that fit their improvement surroundings and SDLC in order to develop secure software.

3.2 : Penetrative Testing

A penetration test, incidentally pen test, is a demonstration performed with a particular objective which confirms the triumph status of the test. A penetration test might be any fusion of attack systems relying upon the objective and guidelines of engagement set. The methodology includes recognizing the target frameworks and the objective, then surveying the data accessible and undertaking any accessible methods concurred upon to achieve the set objective. A penetration test is typically white box where all foundation and framework data is given or black box where just fundamental or no data is furnished other than the organization name. A penetration test will prompt if a framework is powerless against attack, if the defences were sufficient and which defences (if any) were crushed in the penetration test.

A penetration might be compared to looking over a rabbit confirmation wall, which must be entire to keep the rabbits out. In reviewing the wall the penetration tester may recognize a solitary opening vast enough for a rabbit (or themselves) to travel through, once the defence is passed, any further audit of that defence may not happen as the penetration tester proceeds onward to the following security control. These methods there may be a few gaps or vulnerabilities in the first line of defence and the penetration tester just recognized the first found as it was a great adventure. This is the place the contrast laid between a vulnerability appraisal and penetration test - the vulnerability evaluation is everything that you may be powerless to, the penetration test is dependent upon if your defence might be vanquished. Security issues uncovered through the penetration test are displayed to the framework's possessor. Powerful penetration tests will few this data with a precise evaluation of the potential effects to the association and framework a reach of specialized and procedural countermeasures to diminish dangers.

3.3. Code Review

Code review is orderly examination (frequently regarded as associate review) of workstation source code. It is planned to discover and fix missteps ignored in the beginning advancement stage, enhancing both the generally speaking nature of programming and the designers' abilities. Reviews are carried out in different structures, for example, pair programming, informal walkthroughs, and formal inspections. Code reviews can frequently discover and evacuate regular vulnerabilities, for example, configuration string adventures, race conditions, memory breaks and cradle floods, accordingly enhancing software security. Typical code review rates are in the ballpark of 150 lines of code for every hour. Investigating and reviewing more than a couple of hundred lines of code for every hour for critical software, (for example, security critical installed software) may be so quick it is not possible discover slips. Industry information shows that code reviews can achieve at most an 85% deformity evacuation rate with a normal rate of something like 65%; the sorts of deformities caught in code reviews have likewise been examined. Taking into account experimental proof it appears that up to 75% of code review imperfections influence software resolvability as opposed to usefulness making code reviews a fantastic instrument for software organizations with long item or system life cycles.

3.4. Metasploit Framework:

The Metasploit Project is a computer security project that gives data about security vulnerabilities and helps in entrance testing and IDS signature improvement. Its best-known sub- project is the open source Metasploit Framework, an apparatus for advancing and executing adventure code against a remote target machine. Other imperative sub-projects might be the Opcode Database, shell code file and related examination. The Metasploit Project is well known for it's against forensic and avoidance apparatuses, some of which are incorporated with the Metasploit Framework.

IV. PROPOSED SYSTEM

The first block of the block diagram represents source code of web application. In our case we are considering Sugar CRM as it is open source web application and its source code is easily available and the whole application is sufficiently large to cover all types of security functionalities. The source code of sugar CRM is in PHP.

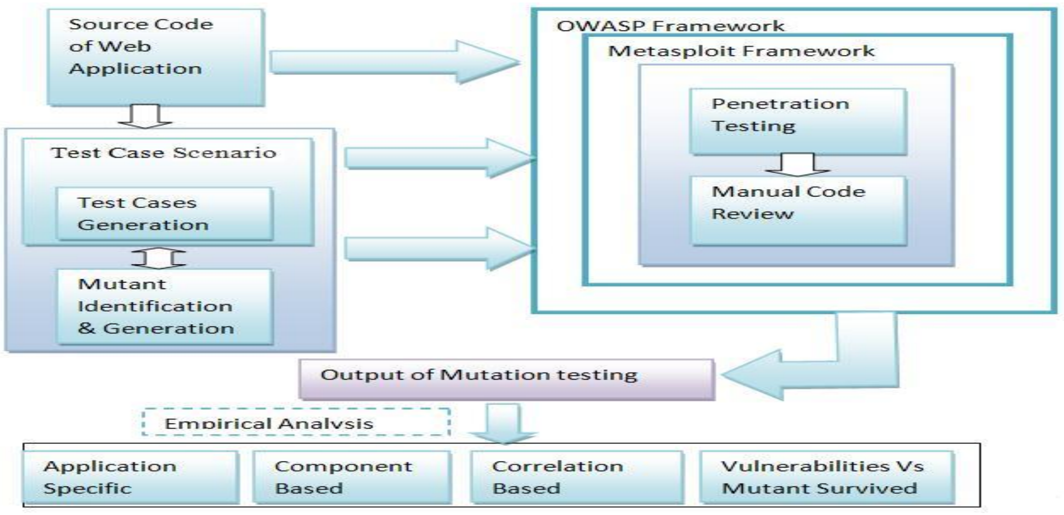


Fig.1 System Architecture

Using the source code we write test cases considering various test case scenarios like code sanitation and covering all code components and all security functionalities. Then mutants are identified using the source code, mutants are generated manually again keeping the security functionalities and vulnerabilities in mind. These mutants are checked against the test cases written if mutants are killed then test cases are sufficient and if mutants are not killed then test cases are again written accordingly. We are using manual approach for mutant creation to avoid the problem of equivalent mutants which is present in case of automated mutant generation by tools. This whole process is iterative and mutation testing helps to write strong test cases. Test cases are written in Java. Now using these test cases we do penetrative testing of source code. In this part we are checking how well software's defences against all types of vulnerabilities leads us to the empirical analysis of this whole process we mainly divide it in four parts. First one is application specific i.e. it looks at security parameters and vulnerabilities a single application is able to avoid. Second empirical analysis is component based it lets us know which software component is more vulnerable to security faults. Third empirical analysis is of correlation between code coverage and mutation score for module M. Last empirical analysis is of OWASP's top ten critical security vulnerabilities v/s Mutants survived. Once all these empirical analysis is done we formulate the analysis data in charts and graphs under various parameters which helps us understand various issues such as which vulnerabilities are to be addressed more critically , how to design test cases for security functionalities , which component of the software is more vulnerable etc.

4.1 Problem Definition

An Empirical evaluation of mutation testing for improving the test security of Web- Application.

4.2. Problem formation

For given software Let,
LCm= Line of code for module m. MGm=Mutants generated for module m.
SIMMs=syntactically incorrect mutants for module m. TMRm=Total mutant retested for module m. MNKm=Mutants not killed for module m. EMm=Evaluate mutants for module m.
MSm=Mutant survived for module m. MSCm=Mutation score for module m. CCm=Code coverage for module m.
SCm=Statement coverage for module m.
Total correlation between two vector samples using mean-square contingency.
Coefficient is given by

$$Ø^2\,(A,B)=\frac{1}{\min(d1,d2)-1}\sum_{i=1}^{d1}\sum_{j=1}^{d2}\frac{(fi,j-fifj)2}{fifj}$$

Where d1 and d2 are sample domain sizes. Mutation Score:
MS(P,T) = DM(P,T)/M(P) – EM(P),
Where DM(P,T) is Number of mutants killed by test set T, M(P) is total number of mutants and EM(P) is number of mutants

$$\text{Coverage} = \frac{\text{Number of Coverage items exercised}}{\text{total no.of coverage items}} \times 100\%$$

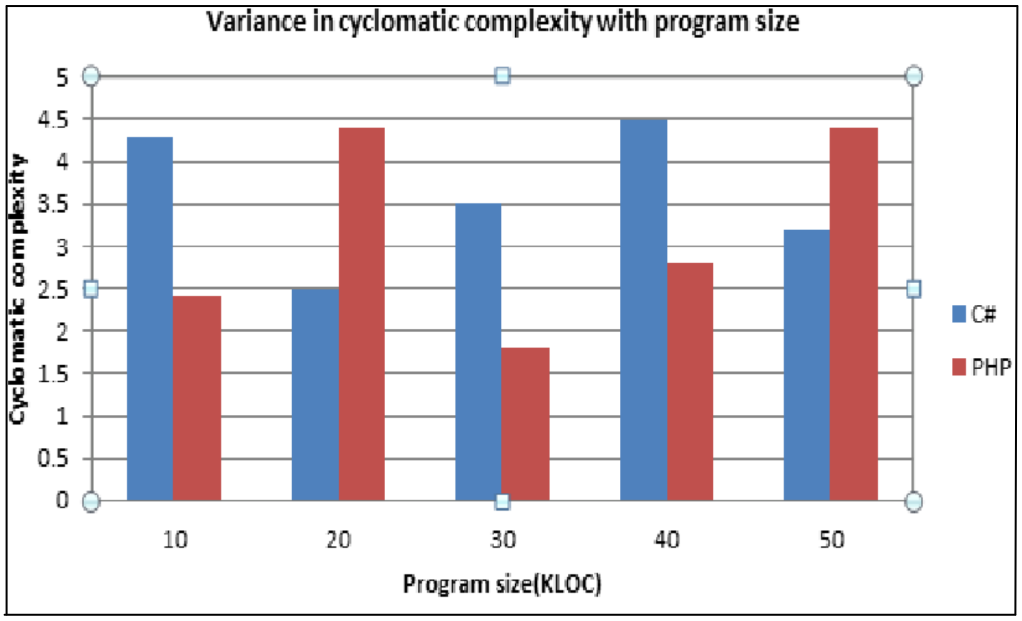
The basic coverage measure is where the coverage item is whatever we have been able to count and see whether a test has exercised. There is danger in using a coverage measure. But, 100% coverage does not mean 100% tested. Coverage techniques measure only one dimension of multidimensional concept. Code coverage should be as high as possible to check all the modules and detect faults. We formulate our problem statement as for given software with m modules and find the values MGm, SIMm,TMRm, MNKm EMm, MSm, MSCm, CCm, SCm after performing mutation testing for security.

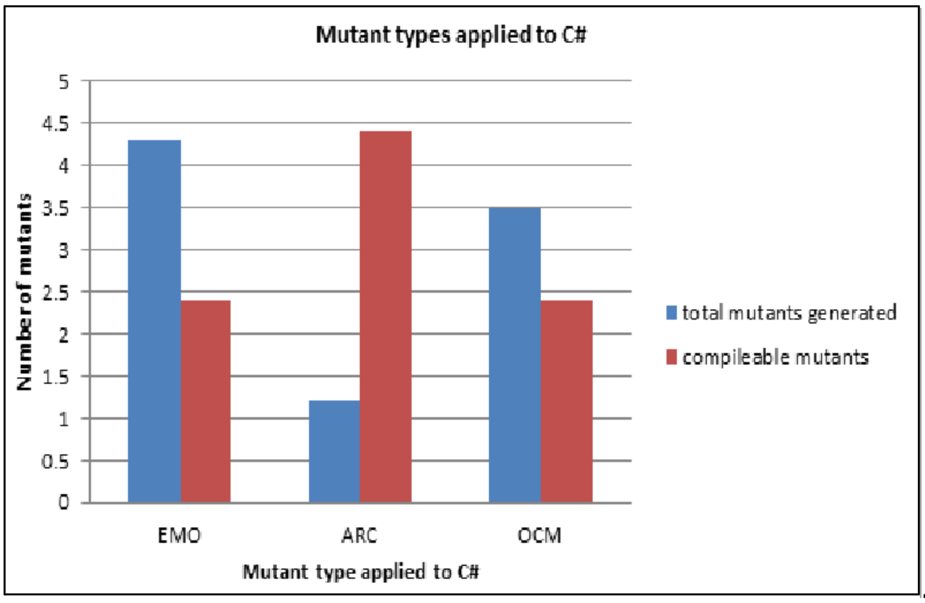
4.3. Algorithm:

Step 1: Identify mutant operators in web programming language like PHP.
Step 2: Consider any web based application and write vulnerability assessment and code review test cases for it.
Step 3: Using mutant operators and test cases in step 2 perform mutation testing.
Step4: Using the new test cases after mutation testing perform penetrative testing on source code.
Step5: Using the results of step 4 manually review the code. Step6: For various module obtain MGm, LCm, SIMm, TMRm, MNKm, EMm, MSm, MSCm values.
Step 7: Using CCm values of various modules perform empirical analysis with respect to other values.
Step 8: Using SCm values of various module perform empirical analysis with respect to other values.
Step 9: Using total correlation between code average CCm and MSCm perform empirical analysis.

4.4 Results

In this subsection we are giving some of the results found in our empirical analysis. The first one is of cyclomatic complexity v/s lines of code which helps us to know the level of mutants i.e. higher order or lower order. The second one is the mutant types that were applied to c#. The data set in our work is source code of web applications written in C# and PHP.





V. CONCLUSIONS

In our approach we tried to empirically evaluate the process of mutation testing giving developer an idea for the future . One of the key security polishes that needs to be set up with specific end goal to relieve the expanding number of vulnerabilities in Web applications, is an organized security testing technique. The way of Web applications requires an iteration furthermore evolutionary methodology to advancement. Hence, the structured security testing approach requirements to have the capacity of being adjusts to such nature's domain, and it should be particular for Web applications. The most connected security testing approaches today are broad and are frequently excessively confused with their numerous exercises and stages.. In this postulation, the creator has demonstrated that by utilizing an organized security testing procedure particularly created for Web applications, expedites an altogether more powerful method for performing security tests on Web applications contrasted with existing specially appointed methods for performing security tests. The components that the creator used to measure the proficiency were: the measure of time used on the security testing process, the measure of vulnerabilities found throughout the security testing procedure and the capacity to moderate false-positives throughout the security testing procedure.

ACKNOWLEDGMENT

I would like to extend my sincere appreciation to my guide Prof.N.D.Kale (Department of Computer Engineering, P.V.P.I.T) for his guidance, encouragement and continuous support throughout the course of this work. I would also like to thank my staff members for their unconditional support and showing interest in my work and giving me their valuable opinions. Last but not the least I would like to thank my family members for their everlasting love, support, and belief in me throughout my life and studies, thanks would never suffice. I hope to make them proud.

REFERENCES

[1] D. Daniels, R. Myers, and A. Hilton, "White Box Software Development," Proc. 11th Safety-Critical Systems Symp., Feb. 2003.

[2] H. Do and G.E Rothermel, "On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques," IEEE Trans. Software Eng., vol. 32, no. 9, pp. 733-752, Aug. 2006.

[3] H. Agrawal, R.A. DeMilo, B.Hathaway,W. Hsu,E.W.Krauser, R.J. Martin, A.P. Mathur and E. Spafford, "Design of mutant Operator for C programming language", Technical report SERC-TR-41P,Purdue, West Lafayette,Ind,Mar.1989.

[4] H. Do and G.E Rothermel, "On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques," IEEE Trans. Software Eng., vol. 32, no. 9, pp. 733-752, Aug. 2006.

[5] J.J. Chilenski, "An Investigation of Three Forms of the Decision Coverage (MCDC) Criterion," Report DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C.,Apr. 2001.G. R. Faulhaber, "Design of service systems with priority reservation," in Conf. Rec. 1995 IEEE Int. Conf. Communications, pp. 3–8.

[6] J.H.Andrews, L.C. Briand and Y. Labiche,"Is Mutation an Approximate Tool for Testing Experiments?" Proc. IEEE Int'I Conf, Software Engg.pp. 402-411,2005.

[7] J.J. Chilenski and S.P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing," Software Eng. J.,vol. 9, no. 5, pp. 193-200, 1994.C. J. Kaufman, Rocky Mountain Research Lab., Boulder, CO, private communication, May 1995.

[8] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practical Programmer," Computer, vol. 11, no. 4, pp. 34-41, Apr. 1978

[9] R. Butler and G. Finelli, "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software," IEEE Trans. Software Eng., vol. 19, no. 1, pp. 3-12, Jan. 1993.