Regression Testing for Trusted Database Applications

Ramzi A. Haraty and Wissam Chehab
Lebanese American University
Division of Computer Science and Mathematics
Beirut, Lebanon
Email: rharaty@lau.edu.lb

Abstract

Regression testing is any type of software testing, which seeks to uncover regression bugs. Regression bugs occur as a consequence of program changes. Common methods of regression testing are re-running previously run tests and checking whether previously-fixed faults have re-emerged. Regression testing must be conducted to confirm that recent program changes have not harmfully affected existing features and new tests must be created to test new features. Testers might rerun all test cases generated at earlier stages to ensure that the program behaves as expected. However, as a program evolves the regression test set grows larger, old tests are rarely discarded, and the expense of regression testing grows. Repeating all previous test cases in regression testing after each major or minor software revision or patch is often impossible due to time pressure and budget constraints. On the other hand, for software revalidation by arbitrarily omitting test cases used in regression testing is risky.

Our proposed algorithms automate an important portion of the regression-testing process, and they operate more efficiently than most other regression test selection algorithms. The algorithms are more general than most other techniques. They handle regression test selection for single procedures and for groups of interacting procedures. They also handle all language constructs and all types of program modifications for procedural languages.

Keywords: database applications, regression testing, and trustworthy.

1. Introduction

Estimates indicate that software maintenance activities account for as much as two-thirds of the cost of software production [39]. One necessary but expensive maintenance task is regression testing, performed on a modified program to introduce confidence that changes thave have been made are correct, and have not adversely affected unchanged portions of the program. An important difference between regression testing and development testing is that during regression testing an established set of tests is available for reuse. One approach to reusing tests, the retest all approach, chooses all such tests, but this strategy may consume excessive time and resources. An alternate approach, selective retest, chooses a subset of tests from the old test set, and uses this subset to test the modified program.

Although many techniques for selective retest have been developed [1, 3, 4, 13, 17, 20, 21 28, 29 37, 40, 41], there is no established basis for evaluation and comparison of these techniques. Classifying selective retest strategies for evaluation and comparison is difficult because distinct philosophies lie behind the existing approaches. Minimization approaches [3, 19, 21] assume that the goal of regression testing is to reestablish satisfaction of some structural coverage criterion, and aim to identify a minimal set of tests that must be rerun to meet that criterion. Coverage approaches [1, 4, 17, 29, 37, 40, 41], like minimization approaches, rely on coverage criteria, but do not require minimization. Instead, they assume that a second but equally important goal of regression testing is to rerun tests that could produce different output, and they use coverage criteria as a guide in selecting such tests.

Safe approaches [10, 13, 20] place less emphasis on coverage criteria, and aim instead to select every test that will cause the modified program to produce different output than the original program.

These philosophies lead selective retest methods to distinctly different results in test selection. Despite these differences, there are identified categories in which selective retest approaches can be compared and evaluated. These categories are inclusiveness, precision, efficiency, generality, and accountability [9].

- Inclusiveness: measures the extent to which a method chooses tests that will cause the modified program to produce different output.
- Precision: measures the ability of a method to avoid choosing tests that will not cause the modified program to produce different output.
- Efficiency: measures the computational cost and automation ability, and thus practicality, of a selective retest approach.
- Generality: measures the ability of a method to handle realistic and diverse language constructs, arbitrarily complex code modifications, and realistic testing applications.
- Accountability measures a method's support for coverage criteria, that is, the extent to which the method can aid in the evaluation of test suite adequacy.

These categories form criteria for evaluation and comparison of selective retest approaches. The rest of the paper is organized as follows: Section 2 presents background information. Section 3 presents our proposed regression testing techniques. Section 4 outlines the support system. Section 5 discusses the empirical results, and section 6 presents the conclusion and further work.

2. Background

Most work on regression testing addresses the following problem: Given program P, its modified version P', and test set T used previously to test P, find a way, making use of T, to gain sufficient confidence in the correctness of P'. Solutions to the problem typically consist of the following steps:

1. Identify the modifications that were made to P. Some approaches assume the availability of a list of modifications, perhaps created by a cooperating editor that tracks the changes applied to P [9]. Other approaches assume that a mapping of code segments in P to their corresponding segments in P' can be obtained using algorithms that perform slicing [43].
2. Select T' included in T, the set of tests to re-execute on P'. This step may make use of the results of step 1, coupled with test history information that records the input, output, and execution history for each test. An execution history for a given test lists the statements or code segments exercised by that test. For example, figure 1 shows test history information for procedure AVG.

```
S1.     count = 0
S2.     fread(fileptr,n)
S3.     while (not EOF) do
S4.       if (n<0)
S5.         return(error)
          Else
S6.         numarray[count]
S7.         count++
```

```
                Endif
S8.             fread(fileptr,n)
              Endwhile
S9.           avg = calcavg(numarray,count) calcavg(numarray,count)
S10           return(avg)
```

Test number	Input	Output	Execution history
T1	empty file	0	S1,S2,S3,S9,S10
T2	-1	Error	Sl,S2,S3,S4,S5
T3	1 2 3	2	S1,S2,S3,S4,S6,S7,S8,S3,...,S9,S10

Figure 1: AVG and its test history information.

3. Retest P' with T', establishing P' correctness with respect to T'. Since we are concerned with testing the correctness of the modified code in P', we retest P' with each Ti Є T'. As tests in T' are rerun, new test history information may be gathered for them.
4. If necessary, create new tests for P'. When T' does not achieve the required coverage of P', new tests are needed. These may include functional tests required by specification changes, and/or structural tests required by coverage criteria.
5. Create T", a new test set history for P'. The new test set includes tests from steps 2 and 4, and old tests that were not selected, provided they remain valid. New test history information is gathered for tests whose histories have changed, if those histories have not yet been recorded.

The five categories described below are the criteria for evaluating selective retest approaches.

Inclusiveness

Inclusiveness measures the extent to which S chooses modification-revealing tests from T for inclusion in T'. We define inclusiveness relative to a particular program, modified program, and test set as follows: suppose T contains n modification-revealing tests, and S (regression testing strategy) selects m of these tests. The inclusiveness of S relative to P, P', and T is the percentage calculated by the expression ((m/n) * 100).

Precision

Precision measures the extent to which a selective retest strategy omits tests that are non-modification revealing. The definition of precision relative to a particular program, modified program and test set, is as follows:

Efficiency

We measure the efficiency of a selective retest method in terms of its space and time requirements. Space efficiency is affected by the test history and program analysis information a strategy must store. Where time is concerned, a selective retest strategy is more economical than a retest-all strategy if the cost of selecting T' is less than the cost of running the tests in T-T' [16]. Thus, efficiency varies with the size of the test set that a method

selects, as well as with the computational cost of that method. Methods for evaluating algorithms are well understood and will not be discussed in this thesis. However, we discuss several factors that must be considered when evaluating the efficiency of selective retest algorithms.

Generality

The generality of a selective retest method is its ability to function in a wide and practical range of situations. Selective retest algorithms should function in the presence of arbitrarily complex code modifications. Also, although we could apply different methods in different settings, we prefer methods that handle all types of language constructs, and large classes of programs. The need for information on program modifications is also a generality issue since requiring knowledge of modifications may impose unreasonable restrictions.

Accountability

Studies suggest that structural test coverage criteria increase the effectiveness of testing [8]. If a program is initially tested with such a criterion, then after modifications it is desirable to confirm that the criterion remains satisfied. Alternatively, if a program is not initially tested using a coverage criterion, it is still possible to apply a criterion at regression test time, ensuring that all new or modified portions of the code have been covered properly [35]. The term accountability to refer to the extent to which a selective retest method promotes the use of structural coverage criteria. Selective retest methods may promote this use by identifying unsatisfied program components, or selecting tests that maximize the coverage achievable. Both coverage and minimization methods facilitate such efforts.

3. The Proposed Security Regression Testing Technique

Observations

One critical necessary maintenance activity, security regression testing, is performed on modified secure interfaces to provide confidence that the software behaves correctly and modifications have not adversely impacted the system's security, in order that the trusted code remains trused.

An important difference between regression testing and development testing is that, during regression testing, an established suite of tests may be available for reuse. One absurd security regression-testing strategy is to reruns all such tests, but this retest-all approach may consume inordinate time and resources. On the other hand, Selective security retest techniques, attempt to reduce the time required to retest a secure program by selectively reusing tests and selectively retesting the modified program. These techniques address two problems:

1. The problem of selecting tests from an existing test suite, and
2. The problem of determining where additional tests may be required.

Both of these problems are important. Our new strategy presents an enhanced regression test selection technique that is specifically tailored for trusted applications. The approach constructs control flow graphs for a secure procedure or program and its modified version and use these graphs to select tests that execute changed code from the original test suite. The new strategy has several advantages over other regular regression test selection techniques. Unlike many techniques, our algorithms select tests that may now execute new or modified statements and tests that formerly executed statements that have been deleted from the original program.

We prove that under certain conditions the algorithms are safe: that is, they select every test from the original test suite that can expose faults in the modified program. Moreover, they are more precise than other safe algorithms because they select fewer such tests than those algorithms. Our algorithms automate an important portion of the regression-testing process, and they operate more efficiently than most other regression test selection algorithms. Finally, our algorithms are more general than most other techniques. They handle regression test selection for single procedures and for groups of interacting procedures. They also handle all language constructs and all types of program modifications for procedural languages. We have implemented our algorithms and conducted empirical studies on several subject programs and modified versions. The results suggest that, in practice, the algorithms can precisely and safely reduce in a significant way the cost of regression testing of a modified program.

Algorithm for Secure Regression Testing

Our algorithm SelectTests, given in figure 2, takes a procedure P, its changed version P', and the test history for P, and returns T', the subset of tests from T that could possibly expose errors if run on P. The algorithm constructs CDG's for P and P', and then calls procedure Compare with the entry nodes E and E' of the two CDG's.

Compare is a recursive procedure. Given any two CDG nodes N and N', Compare method marks these nodes "visited", and then determines whether the children of these nodes are equivalent. If any two children are not equivalent, a difference between P and P' has been encountered. In this case, the only tests of P that may have traversed the change in P are those that traversed N in P. Thus, Compare returns all tests known to have traversed N. If, on the other hand, the children of N and N' are equivalent, Compare calls itself on all pairs of equivalent non-visited predicate or region nodes that are children of N and N', and returns the union of the tests (if any) required to test changes under these children.

```
Algorithm SelectTests
Input       Procedure P, changed version P', and test set T
Output      test set T'
Begin
      Construct CDG and CDG', CDG's of P and P' let E and E' be entry nodes of CDG and
      CDG' T' = Compare (E, E')
End
Procedure Compare
Input       N and N': nodes in CDG and CDG' output   test set T'
Begin
    Mark N and N' "visited"
    If the children of N and N' differ return (all tests attached to N) else
        T' = NULL
        For each region or predicate child node of N not yet "visited" do
            Find C', the corresponding child of N' T' = T' U Compare(C, C')
        End (* for *) end (* if *) end
```

Figure 2: The SelectTests Algorithm.

Example of Test Selection Using SelectTests

Let us consider procedure MedRec' a shown in figure 3, a changed version of procedure MedRec. In MedRec', statement S7 has mistakenly been deleted, and statement S5a has been added. When called with MedRec and MedRec', SelectTests constructs the CDG's for MedRec and MedRec', and calls procedure compare with entry and entry'. Procedure

Compare finds the children of these nodes equivalent, and invokes itself (invocation 2) on R1 and R1'. Recursive calls continue in this manner on nodes P3 and P3' (invocation 3), R2 and R2' (invocation 4), and P4 and P4' (invocation 5). In each case the children of the nodes are found equivalent.

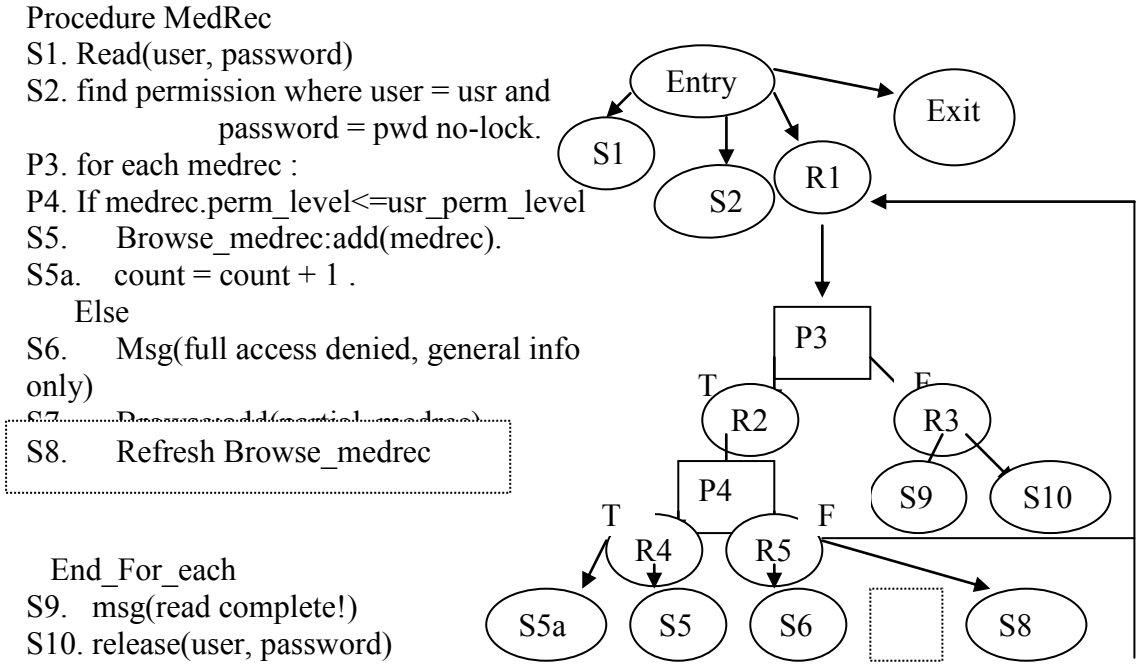

Figure 3: The MedRec' Procedure.

In invocation 6 (on R4 and R4'), procedure Compare discovers nonequivalent children, and thus returns test T2, the only test attached to R4, to invocation 5. Next, Compare calls itself with R5 and R5'. Compare discovers nonequivalent children again, and returns T3, the only test attached to R5. (Because R1 has already been visited, Compare does not examine it again.) Returning up the tree, {T2, T3} is passed back to invocation 4, and then to invocation 3. Here, Compare calls itself with R3 and R3', finds no differences, and returns a null set. Invocation 3 passes {T2, T3} up the tree to invocation 2, then to invocation 1, and finally to the main procedure. The resulting test set, {T2, T3}, contains all the tests that could possibly exhibit different behavior in A2. If the deletion of S7 had been the only change, only {T3} would have been returned. Had the addition of S5a been the only change, only {T2} would have been returned. Most other methods [1, 19, 21, 28, 29, 32, 37, 41] fail to identify T2 and/or T3 as necessary.

To see how SelectTests handles predicate changes, imagine what happens if line P4 in procedure A1 is also changed (erroneously) to "n>0". This change alters only the text associated with node P4 in program A2's CDG. Called with procedures A1 and A2, SelectTests proceeds as in the previous example until it reaches R2 and R2'. Here it finds non equivalent children, and returns {T2, T3}. Note that in this case, no analysis is needed on nodes under P4. No other methods for test case selection make use of the opportunities afforded by the nesting of control dependencies to reduce analysis in this fashion.

Procedure Compare in SelectTests requires a method for determining when the children of two CDG nodes differ. A simple algorithm for doing this checks corresponding nodes for identical text contents.

This simple algorithm is inexpensive and easy to implement, but can be imprecise. Consider, for example, the program fragments shown in Figure 3, in which two unrelated assignment statements, S1 and S2, are swapped. The simple algorithm considers statement order significant, so it finds the children of entry and entry' different, and returns all the tests attached to entry. An algorithm that distinguishes between semantic and syntactic changes would note that the behavior of the nodes under entry and entry' is not, in fact, different, and would continue searching the CDG's for real differences.

This simple algorithm is inexpensive and easy to implement, but can be imprecise. Consider, for example, a secure program fragment, in which two unrelated assignment statements, S1 and S2, are swapped. The simple algorithm considers statement order significant, so it finds the children of entry and entry' different, and returns all the tests attached to entry. So, an algorithm that distinguishes between semantic and syntactic changes would note that the behavior of the nodes under entry and entry' is not, in fact, different, and would continue searching the CDG's for real differences.

On the other hand, any algorithm that is useful in this matter will increase the precision of the approach for an exchange with computational cost. We observe that when the backward slices of two PDG nodes are equal then the statements are semantically equivalent [35]. So, in addition to comparing ordered CDG nodes for textual equivalence, we should compare nodes that are not textually equivalent with backward slicing. Note that the use of backward slicing for nodes is only applicable for "statement nodes" within same regions in both CDS's, since any swapping of positions of "statement nodes" that removes the node from outside its original parent region will result a change in the structure of CDS. Hence, backward slicing is not applicable.

It is necessary to be able to determine when different secure program statements have equivalent behaviors. Given program points p1 and p2 in programs P1 and P2, respectively, we say that p1 and p2 have equivalent behavior if for every initial state on which both P1 and P2 terminate normally; p1 and p2 produce the same sequence of values [5, 43].

Furthermore we use a table or a hash table to store slices of each node that is a child of node N in P and N' in P' who aren't identical and then we attempt to match the slices of children of node N' in P'. We can further improve the hash table approach. We calculate and stores complete slices on each non-identical node in the PDG, we reduce slice calculation by summarizing the slices computed for nodes higher in the hierarchy, and using these summaries in subsequent slice computations. For example, given a PDG, when we encounter and attempt to slice back on a node Si, slices of nodes Si-n have been previously computed and found equivalent. We need slice no farther back from these nodes

4. Support System

We have implemented a security regression testing tool as a support system for this thesis. The objective of the support system is to prove the validity and applicability of the concepts and strategies presented earlier. The developed system helps testers and application maintainer understand the secure applications, identify code changes, support software and requirements updates, enhance, and detect change effects. It helps create a testing environment to select test cases to be rerun when a change is made to the trused application using our 3-phase regression testing methodology.

The system tool is made for Progress datatbase applications programmed using the Progress language.

5. Empirical Results

We use a prototype of a grant-revoke application. We propose a random number of modifications to the application. Then, we study modifications using our maintenance tool and report the regions and the test cases that should be rerun according to the regression testing strategy implemented in the tool. The experimental work is done on a PC, running Pentium IV 3.2 GHz, 512 MB RAM, and using the PC version of Progress.

The application is a grant-revoke secure application (figure 4), which contains most of the language constructs, statement, and controls that we have studied.

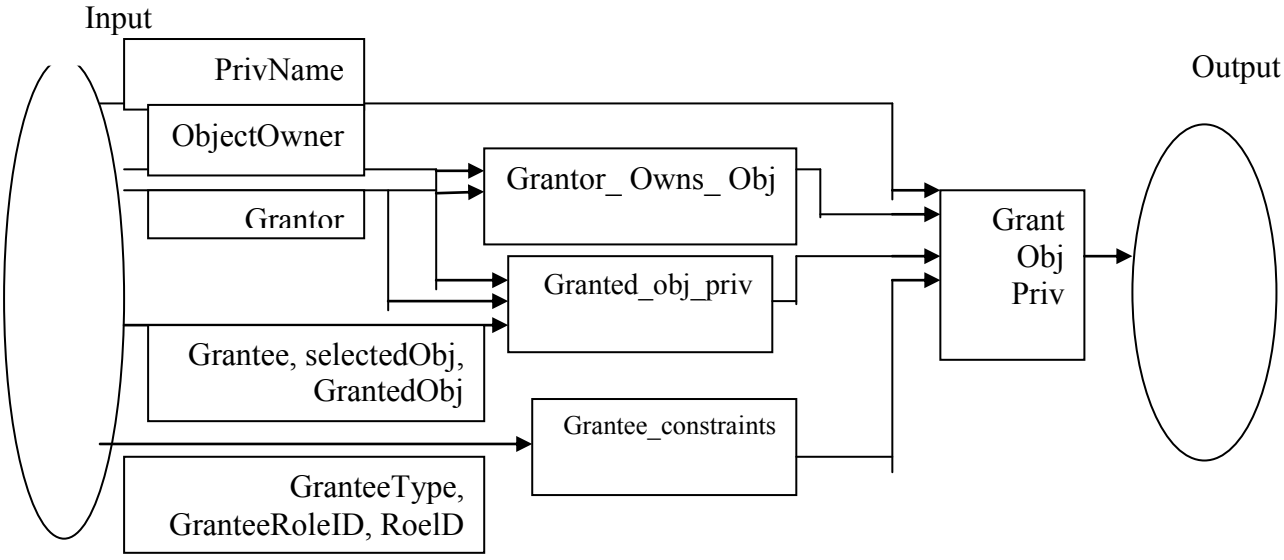

Figure 4: Grant-revoke trusted model application.

In figure 5, we present a summary of test cases presented in this section. We classify these results into two parts. In the first part, we give the results of phase one of our regression testing methodology for secure applications. In the second part, we give the results of phase two, which include a count of test case classes selected by our tool.

Phase 1 results include a list of the following:

1. Directly affected regions.
2. Indirectly affected regions.

Phase 2 results include a list of the following:

1. Test case classes selected by our strategy.
2. Percentage of test case reduction.

Modification Cases	**Directly affected regions**	**Indirectly affected regions**	**Percentage Of test Case selections**	**Percenatage Of Reduction**
1.Modify statement.	26	10	27	72.7
2. Add statement.	6	5	16.5	83.25
3. Delete Statement.	20	9	33.16	67

4. Move Statement.	14	1	30.5	72.25
5.Modify Predicate.	16	7	33	66
6. Delete predicate.	9	7	22	88
7. Add Predicate.	16	14	65.37	33.5
8. Move predicate.	30	17	87	13

Figure 5: Summary of Results. (Total Test cases = 40)

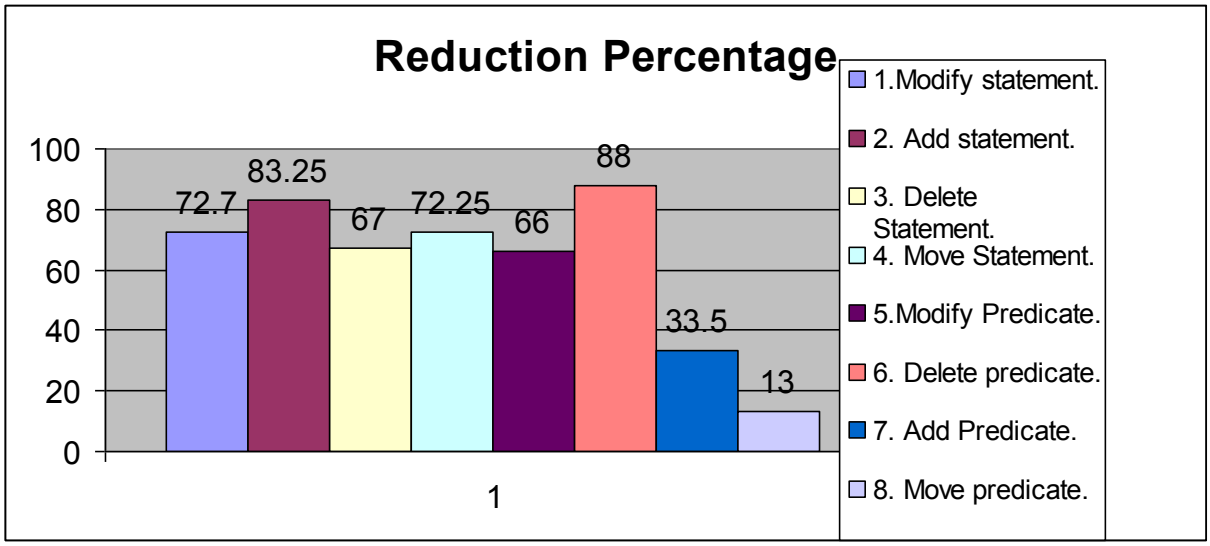

Figure 6 Percentage of test case reduction.

Using our new strategy in trusted regression testing, the tool did a good test reduction and selection job. Out of 80 test vectors of the original test suite used to test the trusted application, we had on average 39% of test cases selected with average of 17 regions directly affected, and 9 regions indirectly affected. This ratio is greatly affected by the number of modifications and the distribution of test cases within the regions. The number of affected regions per modifications depends on the interaction level between the regions in the trused application. On the other hand, Execution time was negligable, and this varies according to the size of the trusted application.

We repeated each experiment five times for each (base trusted program, modified version). The experminetal results showed that our strategy reduced the size of selected tests, and the overall savings were promising.

In fact, our tool reduced test case by more than 60 % on average comparing to "select-all" approach. On the other hand, 60% reduction of test cases is equal to days, hours, even weeks of testing effort. These results show that our approach is precise, and directed towards safety, and greater precision in regression testing of trusted applications.

6. Conclusion and Future Work

In this paper we presented regression testing algorithms for trusted database applications that are efficient and more general than other techniuqes. The algorithms handle regression test selection for signle procedures and groups of interacting procedures. They handle language constructs and different types of program modifications for procedural languages.

Future work includes using more components for case studies, performing additional empirical results to evaluate the effectiveness of our technique, and applying a variety of code

changes to our tool in a production re-test environment. We are looking into utilizing statistical analysis tool such as SPSS to aid in determining the effectiveness of our technique in practice.

Below we briefly expand on some of these issues:

- Implement the technique in production test environment: compared to computer processing time, the time of software engineers is much more costly. Hence, automation is essential to the usability of security regression testing methodology. Our tool for trusted application regression testing can be implemented in a production re-test environment.

- Use statistical analysis tools to determine the effectiveness of our strategy in practice: when we evaluated our technique, we found that the future use of a statistical analysis tool can be powerful in measuring test selections, and improvements on test selections.

From our experience, incomplete and out-of date documentation exists throughout project development and always causes big problems. Because requirement capturing and system design are often done in an informal way, requirement and design documentation is written manually. That causes more serious problems in trusted application re-testing since the document is error-prone. As a part of testing, documentation testing is not done efficiently. Formal regression testing methodologies still have practical problems and not tailored specifically for trusted applications. This thesis has opened new research topics related to secure application regression testing.

References

[1] A.B. Taha, S.M. Thebaut, and S.S. Liu, "An approach to software fault localization and revalidation based on incremental data flow analysis," Proceedings of the 13th Annual International Computer Software and Applications Conference, pp. 527-34, September, 1989.

[2] B. Korel, "The program dependence graph in static program testing," Information Processing Letters, vol. 24, pp. 103-108, January 1987.

[3] B. Sherlund and B. Korel, "Modification oriented software testing," Conference Proceedings: Quality Week 1991, pp. 1-17, 1991.

[4] Benedusi, A. Cimitile, and U. De Carlini. Postmaintenance testing based on path change analysis. In Proceedings of the Conference on Software Maintenance - 1988, pages 352-61, October 1988.

[5] Binkley, "Using semantic differencing to reduce the cost of regression testing," Proceedings of the Conference on Software Maintenance '92, pp. 41-50, November 1992.

[6] E. Duesterwald, R. Gupta and M. L. Soffa, "Rigorous data flow testing through output influences," Proceedings of the 2nd Irvine Software Symposium (ISS'92), pp. 131-145, March 1992.

[7] E. Schatz and B. G. Ryder, "Directed tracing to detect race conditions," LCSR-TR-176, Laboratory for Computer Science Research, Rutgers University, February 1992.

[8] E.F. Miller. Exploitation of software test technology. In Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA), page 159, June 1993.

[9] G. Rothermel and M.J. Harrold, "A comparison of regression test selection techniques" Technical Report 114, Clemson University, Clemson, SC, April 1993.

[10] G. Rothermel and M.J. Harrold, "A safe, efficient regression test selection technique" ACM transactions on software engeneering and methodology, Vol. 6, No. 2 , April 1997.

[11] Gupta and M. L. Soffa, "Automatic generation of a compact test suite," Proceedings of the Twelfth IFIP.
[12] H. Agrawal and J. Horgan, "Dynamic program slicing," Proceedings of ACM SIGPLAN '90 Symposium on Programming Language Design and Implementation, pp. 246-256, June 1990.
[13] H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental Regression Testing. In Proceedings of the Conference on Software Maintenance - 1993, pages 348-357, September 1993.
[14] H. Agrawal, R. DeMillo and E. Spafford, "Dynamic slicing in the presence of unconstrained pointers," Proceedings of the Symposium on Testing, Analysis and Verification, pp.60-73, October 1991.
[15] H. Pande, B. G. Ryder, and W. Landi. Interprocedural def-use associations in C programs. In Proceedings of the Fourth ACM Symposium on Testing, Analysis and Verification (TAV4), October 1991.
[16] H.K.N. Leung and L.J. White, "A cost model to compare regression test strategies," Proceedings of the Conference on Software Maintenance, 1991, pp. 201-8, October, 1991.
[17] H.K.N. Leung and L.J. White, "A study of integration testing and software regression at the integration level." Proceedings of the Conference on Software Maintenance, 1990, pp. 290-300, November, 1990.
[18] J Ferrante, K. J. Ottenstein and J. D. Warren, "The program dependence graph and its use in optimization," ACM Transactions on Programming Languages and Systems, vol. 9, no. 3, pp. 319-349, July 1987.
[19] J. Hartmann and D.J. Robson, "Techniques for selective revalidation," IEEE Software, Vol. 16(1), pp. 31-8, January, 1990.
[20] J. Laski and W. Szemer. Identification of program modifications and its applications in software maintenance. In proceedings of the conference on software maintenance – 1992, apges 282-90, November 1992.
[21] K.F. Fischer, F. Raji, and A. Chruscicki, "A methodology for retesting modified software," Proceedings of the National Telecommunications Conference, Vol. B-6-3, pp. 1-6, November, 1981.
[22] L.J. White and H.K.N. Leung, "A firewall concept for both control-flow and data-flow in regression integration testing," Proceedings of the Conference on Software Maintenance, 1992, pp. 262-70, November, 1992.
[23] M. Davis and E. Weyuker. Computability, Complexity, and Languages. Academic Press, Boston, MA, 1993.
[24] M.J. Harrold and B. A. Malloy, "A unified interprocedural program representation for a maintenance environment," IEEE Transactions on Software Engineering, to appear.
[25] M.J. Harrold and B. A. Malloy, "Data flow testing of parallelized code," Proceedings of the Conference on Software Maintenance '92, pp. 272281, November 1992.
[26] M.J. Harrold and B. A. Malloy, "Performing data flow analysis on the PDG", Technical Report 92-108, Clemson University, March 1992.
[27] M.J. Harrold, B. A. Malloy and G. Rothermel, "Efficient construction of program dependence graphs," Technical Report 92-128 Clemson University, December 1992.
[28] M.J. Harrold and M.L. Soffa, "An incremental approach to unit testing during maintenance," Proceedings of the Conference on Software Maintenance, 1988, pp. 362-7, October, 1988.
[29] M.J. Harrold and M.L. Soffa, "Interprocedural data flow testing," Proceedings of the Third Testing, Analysis and Verification Symposium, pp. 158-67, December, 1989.
[30] M.J. Harrold, B.A. Malloy, and G. Rothermel, "Efficient construction of program dependence graphs," Proceedings of the International Symposium on Software Testing and Analysis 93 (ISSTA93), pp. 160-70, June, 1993.

[31] M. Weiser, "Program slicing," IEEE Transactions on Software Engineering, vol. 5E-10, no. 4, pp. 352-357, July 1884.
[32] P. Benedusi, A. Cimitile, and U. De Carlini, "Postmaintenance testing based on path change analysis," Proceedings of the Conference on Software Maintenance, 1988, pp. 352-61, October, 1988.
[33] R. Ballance and B. Maccabe, "Program dependence graphs for the rest of us," Technical Report, University of New Mexico, November 1992.
[34] R. Cytron, J. Ferrante, B. Rosen and M. Wegman, "Efficiently computing static single assignment form and the control dependence graph," ACM Transactions on Programming Languages and Systems, vol. 13, no. 4, pp. 451-490, October 1991.
[35] R. Gupta, M. J. Harrold and M. L. Soffa, "An approach to regression testing using slicing", Proceedings of the Conference on Software Maintenance '92,pp. 299-308, November 1992.
[36] R. M. Stallman, "Using and porting GNU CC," Free Software Foundation, Inc., Cambridge MA, pp. 73-77, February 1990.
[37] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," Annual ACM Symposium on Principles of Programming Languages, January, 1993.
[38] S. Horwitz, T. Reps and D. Binkley, "Interprocedural slicing using dependence graphs," ACM Transactions on Programming Languages and Systems, v. 12, no. 1, pp. 26-60, January 1990.
[39] Schach, Software Engineering, Aksen Associates, Boston, MA, 1990.
[40] S.S. Yau and Z. Kishimoto, "A method for revalidating modified programs in the maintenance phase," COMPSAC '87: the Eleventh Annual International Computer Software and Applications Conference, pp. 272-7, October, 1987.
[41] T.J. Ostrand and E.J. Weyuker, "Using dataflow analysis for regression testing," Sixth Annual Pacific Northwest Software Quality Conference, pp. 233-47, September, 1988.
[42] U. Linnenkugel and M. Mullerburg, "Test data selection criteria for (software) integration testing," Systems Integration '90. Proceedings of the First International Conference on Systems Integration, pp. 709-17, April, 1990.
[43] W. Yang, "Identifying syntactic differences between two programs," Software-Practice and Experience, Vol. 21(7), pp. 739-55, July, 1991.