Security Functional Testing Using An Interface-Driven Model-Based Test Automation Approach

Ramaswamy Chandramouli
Computer Security Division, NIST
mouli@nist.gov

Mark Blackburn
T-VEC Technologies
blackbur@software.org

Abstract

Independent security functional testing on a product occupies a backseat in traditional security evaluation because of the cost and stringent coverage requirements. In this paper we present the details of an approach we have developed to automate security functional testing. The underlying framework is called TAF (Test Automation Framework) and the toolkit we have developed based on TAF is the TAF-SFT toolkit. The TAF-SFT toolkit uses text-based specifications of security functions provided by the product vendor and the requirements of the underlying security model to develop a machine-readable specification of security functions using the SCR (Software Cost Reduction) formal language. The resultant behavioral specification model is then processed through the TAF-SFT Toolkit to generate test vectors. The behavioral model and the test vectors are then combined with product interface specifications to automatically generate test drivers (test execution code). We illustrate the application of TAF-SFT toolkit for security functional testing of a commercial DBMS product. We also discuss the advantages and disadvantages of using TAF-SFT toolkit for security functional testing and the scenarios under which the impact of disadvantage can be minimized.

1. Introduction

Independent security functional testing (or in general security testing) often occupies a backseat in traditional security evaluations of many commercial products, except in the case of high-assurance products deployed in life-critical environments. The reasons for this scenario are:

(a) Cost – not many security evaluations are performed by evaluators to amortize the initial investment in developing the infrastructure to perform security testing as well as the non-reusability of the previously developed tests and

(b) Technical Complexity – this arises from the complexity of representing the security function specifications and the coverage requirements for the test data used for conducting the tests.

In this paper we describe an approach and an associated toolkit that addresses the issues outlined above in the case of security functional testing. The underlying framework of our approach is called the Test Automation Framework (TAF)[*] [1,2]. The TAF is an architectural framework that automates the process of system or software testing by providing end-to-end tool support for the various process steps. These process steps include functional model development, model analysis, automated test code generation, automated test execution and results analysis. We have developed a toolkit called the TAF-SFT toolkit that applies TAF to security functional testing of a product and have demonstrated the application of this toolkit for testing the security behavior of a commercial DBMS product.

The organization of the rest of our paper is as follows: In section 2 we discuss the characteristics of security functional testing and the role it plays in the overall realm of security testing. In section 3 we describe the details of the application of TAF to security functional testing (as well as the TAF-SFT toolkit) in terms of the functions performed in the various TAF process steps. Section 4 illustrates the application of TAF-SFT toolkit for security functional testing of the Oracle DBMS product using a sample set of security function specifications. Section 5 discusses the advantages and disadvantages of our automated security functional testing approach and conditions under which the impact of disadvantages can be minimized.

[*] Certain commercial products and standards are mentioned in this paper. This does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the products and standards mentioned are necessarily the best available for the purpose.

2. Characteristics of Security Functional Testing

There are subtle differences between traditional software conformance testing and security testing in general, in terms of purpose, scope, emphasis, error implications and strategy [3]. The main purpose of software conformance testing is verification of *correctness* of implementations with respect to specifications. The market largely determines the *effectiveness* of the implementation. However security testing is concerned with both correctness and effectiveness since measures of effectiveness such as strength of functions and robustness are very much an integral part of any security specifications. In traditional conformance testing, the emphasis is on testing the implementation for conformance to functional specifications while in security testing the product must be tested not only for conformance to security function specifications but also for compliance with mandatory features of the underlying security model. For example testing an access control function in a DBMS product will involve not only verification of specified behavior (correct access denials and clearances for a particular user) but also conformance to the underlying Discretionary Access Control Model (DAC) that provides the logic governing denials and clearances depending upon certain user attributes and state variables. In traditional conformance testing, verification using test cases that satisfy some statistical coverage measures can provide the assurance that certain defects will seldom occur. However in security testing, complete test coverage is required since obscure flaws can be exploited individually and collectively to subvert the behavior of other correctly implemented functions. The requirement for complete coverage can result in the number of test cases for security testing being an order of magnitude more than for traditional conformance testing.

Security Testing itself can be generally classified as security functional testing and security vulnerability testing. **Security functional testing** involves testing the product or implementation for conformance to the security function specifications as well as for the underlying security model. The conformance criteria state the conditions necessary for the product to exhibit the desired security behavior or satisfy a security property. In other words security functional testing involves what the product *should do*. **Security vulnerability testing** on the other hand is concerned with identification of flaws in design or implementation that may be exploited to subvert the security behavior which has been made possible by the correct implementation of the security functions. In other words security vulnerability testing involves testing the product for what it *should not do*.

3. TAF for Security Functional Testing

Our application of the TAF to security functional testing (and the associated Toolkit TAF-SFT) involves the use of the SCR formal language [4], its support tool TTM and test automation tool T-VEC [5,6], and the following process steps:

TAF-SFT (Step 1): Develop a behavioral model of the security functions specifications of a product (as obtained from its text-based documentation) using a tabular-type specification called SCR model using the TTM tool.

TAF-SFT (Step 2): Translate SCR specifications into T-VEC specifications using a model translator.

TAF-SFT (Step 3): Generate test vectors from the transformed SCR specification

TAF-SFT (Step 4): Develop test driver schemas and object mappings (explained later) for target test environment.

TAF-SFT (Step 5): Generate test drivers, execute tests and generate test report.

The process flow diagram of the above steps is given in Figure 3.1 and the details of each of the above process steps are explained in the following sections.

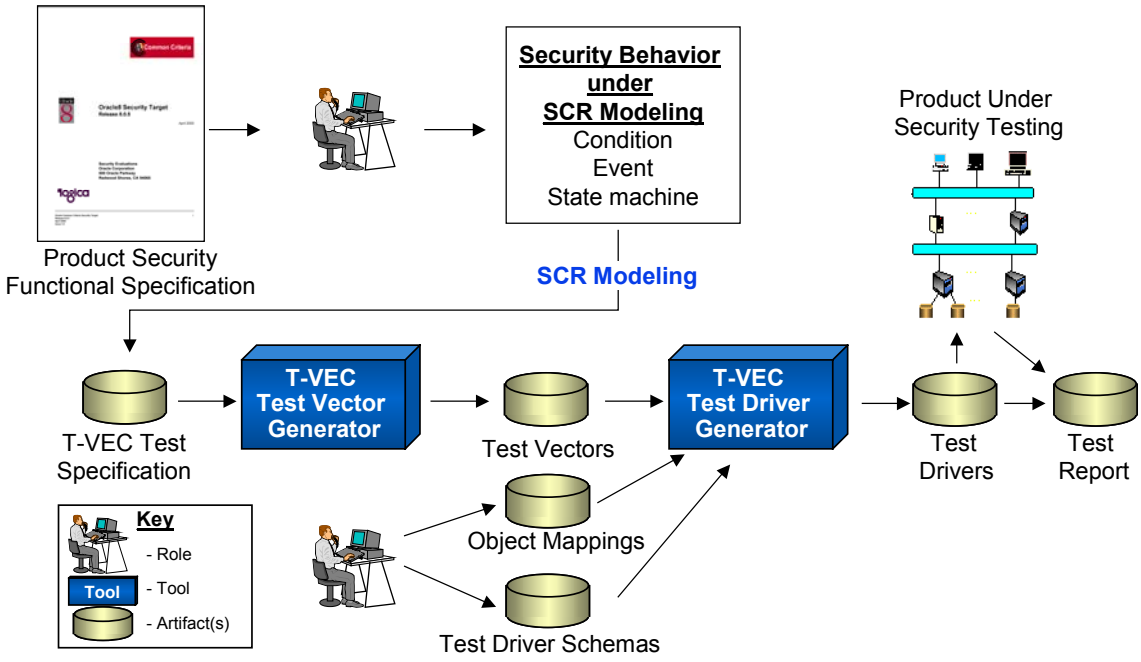

Figure 3.1
(Test Automation Framework for Security Functional Testing – Process Flow Diagram)

3.1 Develop SCR Model of Security Function specifications (TAF-SFT (Step 1))

The SCR models the behavior of a software system in the form of "a set of functions associated with output variables (or controlled variables)" [7]. These functions will involve besides controlled variables, other variables such as monitored variables, terms and mode classes. A monitored variable represents an input quantity (input variable). A term is an auxiliary variable (that may be a combination of monitored variables or other terms used for simplifying the model or for representing some intermediate concepts). A mode class is a special case of a term whose values are modes. A mode stands for a particular system state.

For example let us consider the modeling of a security function that determines the conditions under which a delete access request can be granted (let us call this the GDA security function). Let us call the controlled (output) variable in this context *Grant_Delete_Access*. The variables determining the value of this output variable (i.e. whether Grant_Delete_Access is TRUE or FALSE) are the particular user (identified by a UserID) and the privileges held by that user. Hence the monitored (input) variables in the context of the GDA security function are *UserID* and *User_Object_Priv*. Let us also define a term variable, User_has_Delete_Access, to group together the combinations of user privileges that are relevant in the context of granting delete access. These variables (and their associated types) and terms are defined and stored in the various dictionaries of the SCR model as follows:

Table 3.1 – Variable Dictionary

Name	Class	Type	Initial Value	Accuracy
UserID	Monitored	UserID_Type	1111	N/A
User_Object_Priv	Monitored	Priv_Type	SELECT	N/A
Grant_Delete_Access	Controlled	Boolean	FALSE	N/A

Table 3.2 – Term Dictionary

Name	Type	Initial Value	Accuracy
User_has_Delete_Access	Boolean	FALSE	N/A

Table 3.3 – Type Dictionary

Name	Base Type	Units	Legal Values
UserID_Type	Integer	N/A	[1111-9999]
Priv_Type	Enumerated	N/A	ALL, DELETE, UPDATE, SELECT, INSERT

Having represented the input, term and output variables involved in modeling the GDA security function, we will show how the behavior of the GDA security function is modeled using SCR functions. The functions in SCR consist of conditions and events and the corresponding tables used to represent them are called 'condition function tables' and 'event function tables' respectively. A condition is a predicate defined on one or more state variables (a state variable is a monitored or controlled variable, a mode class or a term). In our GDA security function context, the conditions that determine the truth values for the term variable User_has_Delete_Access are represented using the following condition function table:

Table 3.4 – Condition Function Table for the term variable – User_has_Delete_Access

Table Name	Condition	
	(User_Object_Priv = 'ALL') OR (User_Object_Priv = 'DELETE')	(User_Obj_Priv != 'ALL') AND (User_Obj_Priv != 'DELETE')
User_has_Delete_Access =	TRUE	FALSE

Coming back to the discussion of conditions and events, an event refers to a moment in time and is said to occur when the value of a condition changes from true to false or vice versa. An example of an event when a user gains delete privileges when he/she did not have that privilege before is represented as:

@T(User_has_Delete_Priv)

For verification of security functions, we are interested in the external behavior of a product under various security conditions and not in the valid security state transitions. Hence in our SCR model of security function specification we will be dealing with only conditions (represented using condition function tables) rather than events (or event function tables). The condition function table for the controlled variable Grant_Delete_Access for our GDA security function is:

Table 3.5 – Condition Function Table for the controlled variable – Grant_Delete_Access

Table Name	Condition	
	(UserID=Active_user) AND (User_has_Delete_Access)	(UserID != Active_User) OR NOT(User_has_Delete_Access)
Grant_Delete_Access	TRUE	FALSE

3.2 Translate SCR specifications to T-VEC (Test Vector) specifications (TAF-SFT (Step 2))

As we have seen through an example specification, the SCR model is composed of tables of conditions and events. This model is not in a form that supports test generation. Hence we use a T-VEC model translator tool to transform the SCR model into a test specification model (called the T-VEC Linear Form). A test specification model is defined by *outputs, inputs, functional relationships* and *relevance predicates*. A functional relationship is an input-output relation and a relevance predicate is a grouping of constraints on the inputs associated with a given input-output relation. For our example SCR model, the functional relationship obtained for our GDA security function will be:

(UserID = Active_User) & User_has_Delete_Access → Grant_delete_access

The relevance predicate associated with the above functional relationship (or input-output relation) is:

((UserID = Active_User) AND (User_Obj_Priv = 'ALL'))

OR

((UserID = Active_User) AND (User_Obj_Priv = 'DELETE'))

From the above expressions it can be seen that a relevance predicate is expressed in disjunctive normal form, i.e. as a set of disjunctions of conjunctions. Each disjunction is referred to as a domain convergence path (DCP).

3.3 Test Vector generation from the transformed SCR specification & Coverage Analysis (TAF-SFT (Step 3))

The T-VEC test generator tool generates test data for subdomains of an input variable space based on the constraints of a DCP. In other words it tries to generate a test vector for each DCP. In fact the test vector is a set of test input values derived from a DCP and an expected output value derived from the input-output relation. Informally, from a test vector generation perspective, a specification is satisfiable if at least one test vector exists for each DCP.

There may exist input variables in the input-output relation that are not constrained by DCP predicates. The test vector also generates additional test points by incorporating boundary value combinations from these unconstrained inputs (e.g. low bound and high bound for numeric objects, sets for enumerated variable). The incorporation of these additional test points helps to prove that unconstrained inputs do not the affect the expected value of the input-output relation.

However the presence of a test vector for each DCP is no guarantee that collectively the set of test vectors is sufficient to verify all the path conditions for a functional relationship. This scenario may result if contradictions exist among DCPs. Hence we used the T-VEC coverage analyzer to detect these contradictions and ensure that the test vectors provide the intended coverage.

3.4 Develop test driver schemas and object mappings for target test environment (TAF-SFT (Step 4))

We now have the translated SCR model containing the behavioral specification of security functions and the associated test vectors. These two documents by themselves do not provide sufficient information to the test driver to generate executable test code in a procedural language. We do need to provide the test driver generator the knowledge of the product's interface API (that pertains to the test code language) and any other relevant APIs needed for extraction of information pertaining to the product's state. This is exactly the information that is provided by the 'Object Mapping' file that is shown in Figure 3.1. More specifically, the 'Object mapping file' provides the mapping between the behavioral model variables and the interface elements needed to set, retrieve or evaluate the values of those model variables. The combination of the behavioral model and the object mapping information is called the 'verification model' since it represents the complete specification required for carrying out the product's functional verification process.

The last but not the least important piece of information that the test driver generator needs is the generic sequence of steps needed for executing any test. It is this piece of information that is provided in the 'Test driver Schema' file. The test driver schema file describes the simple algorithmic pattern that is used to load, execute and receive test data and other environmental information pertaining to the target test environment.

3.5 Generate test drivers, execute tests and generate test results report (TAF-SFT (Step 5))

The test driver generator operates on the behavioral model, test vectors, object mapping information and test execution template definitions (in the test driver schema file) to generate the executable test code. In our TAF-SFT toolkit, the test driver generator generates code in Java, though conceptually any language generator module can be incorporated within the test driver generator. The test driver generator also generates the 'Expected Outputs File' whose format is again specified in the test driver schema.

The generated test driver code is then executed against the product by incorporation of the appropriate run-time libraries (e.g. Java Virtual Machine and java run-time libraries). This process generates the 'Actual Outputs' File. The last process in our TAF application for security functional testing is the 'Cross Comparison' that compares the expected outputs with the actual outputs to generate the test results report.

4. TAF-SFT Toolkit for a commercial DBMS product

We will now illustrate the application of the TAF-SFT Toolkit for security functional testing of a commercial DBMS product – Oracle 8.0.5. The generic process flow steps in TAF-SFT (sections 3.1 through 3.5) mapped to the application of TAF-SFT for the Oracle DBMS product are given in Figure 4.1. As stated in section 3.1, our first step is to obtain the security functions specifications. We obtained the text-based specification of the security functions for Oracle 8.0.5 from the Oracle 8.0.5 Security Target (ST) Document [8]. The Security Target is a structured specification of security functional requirements as well as specification of security functions that meet those requirements expressed using a pre-defined catalog of requirements and function representations in the international security criteria ISO/IS 15408 [9].

The next step after obtaining the text-based security functions specifications is to develop an SCR model of these specifications. In the next subsection we illustrate the development of SCR model specification for an example security function specification for the Oracle DBMS.

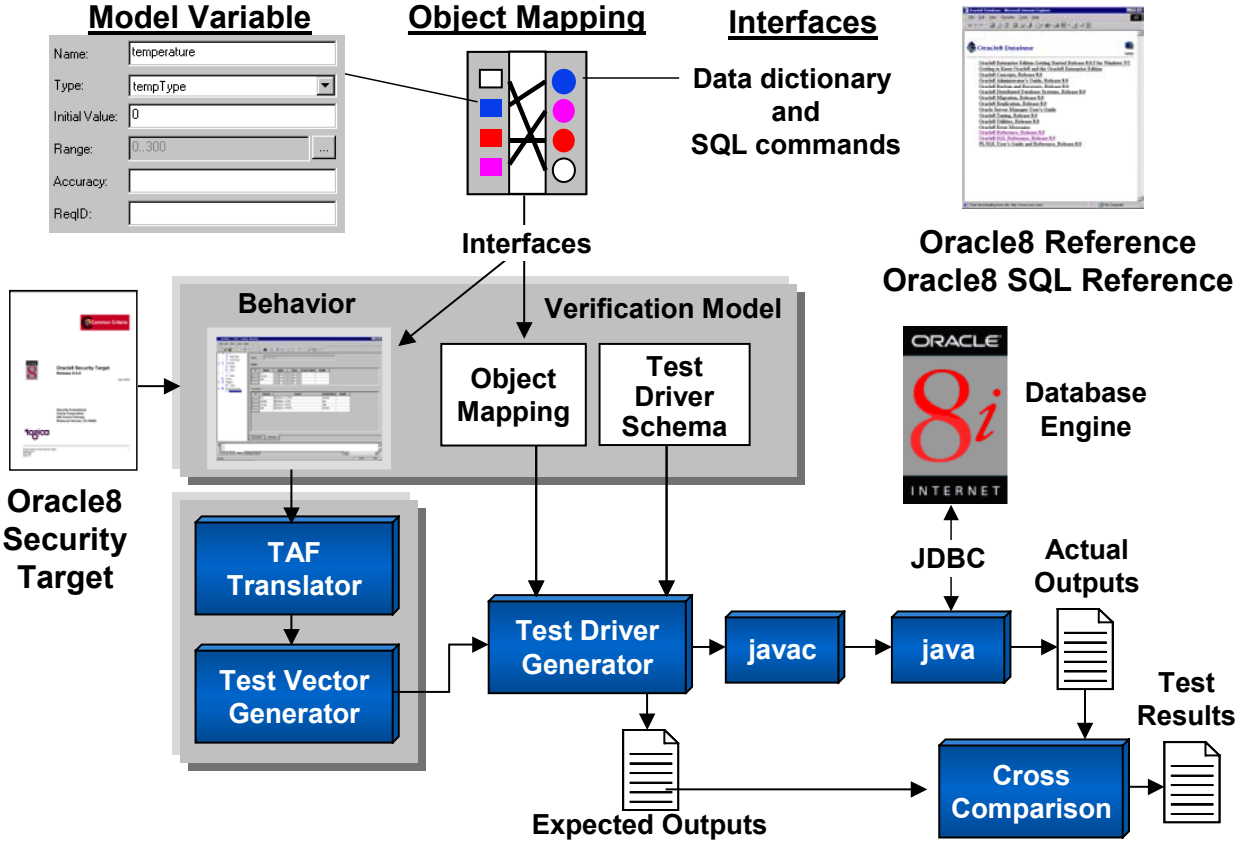

Figure 4.1 – Application of TAF-SFT Toolkit for Oracle DBMS Security Functional Testing

4.1 SCR Model of an Oracle DBMS Security Function Specification

The specification for a security function that stipulates the conditions under which an Oracle database user can grant an object privilege to another user as stated in the Oracle ST document is:

Granting Object Privilege Capability (GOP) - A normal user (the grantor) can grant an object privilege to another user, role or PUBLIC (the grantee) only if:
GOP (a): the grantor is the owner of the object ; or
GOP(b): the grantor has been granted the object privilege with the GRANT OPTION.
A role represents a group of privileges associated with a business process. The keyword PUBLIC represents all users.

Recall that the formulation of an SCR model requires the identification of variables. The various variables identified for modeling the GOP security function are:

(a) Monitored Variables (input variable) - grantor, grantee, selectedObj, selectedObjPriv, granteeType, grantedObj, grantedObjPriv

(b) Controlled Variable (output variable) – grant_obj_priv_OK – A Boolean variable that will have the value TRUE when the conditions for 'granting objective privilege' by one user to another are satisfied.

In addition to the above variables, we need two term variables to complete the GOP function specification in SCR. They are: (a) grantor_owns_object (to incorporate the conditions that affirm the fact that the grantor is the owner of the selected object – the requirement GOP(a)) and (b) has_grantable_obj_privs (to incorporate the conditions that affirm that the grantor holds the privilege in question for the selected object with the ability to propagate (GRANT OPTION) – the requirement GOP(b)). Expressing the conditions that affirm the truth-values for the above discussed term variables in SCR notation we get:

grantor_owns_object – TRUE when grantor = selectedObjOwner (4.1.1)

has_grantable_obj_privs – TRUE when selectedObj = grantedObj AND selectedObjPriv = grantedObjPriv AND GRANT_OPTION (4.1.2)

Based on our previous discussion, it should be clear that our security functional testing involves not only testing the security function specifications, but also the underlying model semantics (in this case the Discretionary Access Control (DAC) model)). Clearly the DAC model semantics in our case is that the object owner and the holder of the privilege (with GRANT option) are two different entities. This DAC model semantic constraint should be added to the term condition 4.1.2 above to yield:

has_grantable_obj_privs – TRUE when
selectedObj = grantedObj AND
selectedObjPriv = grantedObjPriv AND
GRANT_OPTION AND
selectedObjOwner != grantor AND
selectedObjOwner != grantee (4.1.2)'

Now that the expressions 4.1.1 and 4.1.2' represents our requirements GOP(a) and GOP(b) (along with DAC model semantics) our SCR condition for the entire GOP function becomes:
grant_obj_priv_OK – TRUE when grantor_owns_object OR has_grantable_obj_privs

.Table 4.1 – SCR Condition Function Tables for the GOP Security Function

Table Name	Condition	
	grantor = selectedObjOwner	NOT(grantor = selectedObjOwner)
grantor_owns_object =	TRUE	FALSE

Table Name	Condition		
	(GRANT_OPTION AND selectedObjPriv = grantedObjPriv) AND selectedObj = grantedObj AND selectedObjOwner != grantor AND selectedObjOwner != grantee	NOT(GRANT_OPTION AND selectedObjPriv = grantedObjPriv) AND selectedObj = grantedObj AND selectedObjOwner != grantor AND selectedObjOwner != grantee	**DAC Constraints**
has_grantable_obj_privs =	TRUE	FALSE	

Table Name	Condition		
	((grantor_owns_object) OR (has_grantable_obj_privs)) AND (grantor != grantee) AND (granteeType = user OR (granteeType = role AND granteeRoleID = valid_roleID) OR granteeType = PUBLIC) AND (selectedObjPriv = ALL OR selectedObjPriv = UPDATE OR selectedObjPriv = SELECT OR selectedObjPriv = INSERT OR selectedObjPriv = DELETE)	(NOT(grantor_owns_object)) AND (NOT(has_grantable_obj_privs)) AND (grantor != grantee) AND (granteeType = user OR (granteeType = role AND granteeRoleID = valid_roleID)) AND (selectedObjPriv = ALL OR selectedObjPriv = UPDATE OR selectedObjPriv = SELECT OR selectedObjPriv = INSERT OR selectedObjPriv = DELETE)	**GOP(a)** **GOP(b)** **Domain Constraints**
grant_obj_priv_OK =	TRUE	FALSE	

Now our SCR specification of the GOP security function fully represents the claimed functionality in the Oracle ST document along with DAC model semantics. However we have still not incorporated constraints that relate directly to the Oracle DBMS domain. These relate to the fact that the grantee can only be of type user, role or PUBLIC and that the object privilege can only be one of UPDATE, DELETE, SELCT, INSERT or ALL (as they are the valid privilege modes for objects managed by the DBMS). Hence these domain constraints should also be incorporated to complete the GOP function specification. The SCR condition tables dealing with the conditions for the term variables (4.1.1 and 4.1.2') as well as for the controlled variable (grant_obj_priv_OK) (including the domain constraints) are given in table 4.1.

4.2 Generation of Test Vectors for DBMS Security Function (GOP)

As outlined in our approach in section 3.2, we next processed our SCR model for ' Granting Object Privilege Capability' (GOP security function) through the T-VEC translator tool to obtain the following functional relationship (input-output relation).

((grantor_owns_object) OR (has_grantable_obj_privs)) AND <domain_constraints> → grant_obj_priv_OK (4.2.1)

The next item we obtain from the T-VEC translator tool are the relevance predicates. Recall that the relevance predicate groups together all the constraints associated with input values and is expressed in the form of disjunctions of conjunctions and that each disjunction is called the Domain Convergence Path (DCP). In the GOP security function specification context, each DCP should therefore contain either the component GOP(a) or GOP(b) in table 4.1 along with each of the possible value associations given in the domain constraints. A few examples of DCPs are:

(grantor_owns_object) AND (grantee='user') AND (selectedObjPriv = 'UPDATE') (4.2.2)
(has_grantable_obj_privs) AND (grantee='PUBLIC') AND (selectedObjPriv = 'SELECT') (4.2.3)

In fact we can compute the total number of test vectors for testing our GOP security function, by calculating the number of DCPs in the relevance predicate and the fact that the test vector generator will generate at least one test vector for a DCP. Since a DCP is one disjunction, each of the ORs in our SCR condition function table 4.1 should participate in a DCP. Since the conditions GOP(a) and GOP(b) are connected with OR, each should give rise to a different DCP. On examining the domain constraints we find that there are three OR s for the expressions involving input variable 'grantee' (three possible values for grantee) and five OR s for the expressions involving the input variable 'selectedObjPriv' (five possible values for selectedObjPriv). Hence the total number of disjunctions or DCPs we will obtain will equal 2*3*5 = 30. There should therefore be a minimum of thirty test vectors for testing the GOP security function (all yielding the value TRUE for the controlled variable grant_obj_priv_OK). Including the test cases for grant_obj_priv_OK being FALSE (by negating at least one predicate in each DCP) and additional test points derived from boundary value combinations of unconstrained input variables like grantor, grantee, grantee_ roleID, the test vector generator generated about 80 test vectors for testing the GOP security function. The test vectors are shown in table 4.2.

Table 4.2 – Test Vectors generated for testing the GOP Security Function

#	TSP	grant_obj _priv_OK	grantor	grantee	grantee Type	grantee RoleID	valid_roleID	selected ObjPriv	objOwner	GRANT_ OPTION	granted ObjPriv	selected Obj	granted Obj
1	1	TRUE	1	2	user	2	2	ALL	1	TRUE	ALL	4	4
2	1	TRUE	4	3	user	1	1	ALL	4	FALSE	SELECT	1	1
3	2	TRUE	1	2	user	2	2	UPDATE	1	TRUE	ALL	4	4
4	2	TRUE	4	3	user	1	1	UPDATE	4	FALSE	SELECT	1	1
5	3	TRUE	1	2	user	2	2	SELECT	1	TRUE	ALL	4	4
6	3	TRUE	4	3	user	1	1	SELECT	4	FALSE	SELECT	1	1
7	4	TRUE	1	2	user	2	2	INSERT	1	TRUE	ALL	4	4
8	4	TRUE	4	3	user	1	1	INSERT	4	FALSE	SELECT	1	1
9	5	TRUE	1	2	user	2	2	DELETE	1	TRUE	ALL	4	4
10	5	TRUE	4	3	user	1	1	DELETE	4	FALSE	SELECT	1	1
▪ ▪ ▪													
77	39	FALSE	1	2	role	1	1	INSERT	3	FALSE	ALL	1	1
78	39	FALSE	4	3	role	2	2	INSERT	2	FALSE	SELECT	4	4
79	40	FALSE	1	2	role	1	1	DELETE	3	FALSE	ALL	1	1
80	40	FALSE	4	3	role	2	2	DELETE	2	FALSE	SELECT	4	4

4.3 Generation of Test Drivers for Oracle DBMS testing

At this stage we have the SCR Model (or a translated variant) of Oracle DBMS's security behavioral specification and associated test vectors. These two documents in themselves are not sufficient to generate executable test code (or test driver) in Java (which is the language capability of test driver generator in our TAF-SFT toolkit) for testing the security functions of the Oracle DBMS product. We do need the knowledge of the Java APIs to interface with Oracle DBMS as well as the knowledge of the structure of the data stores that contain the security state information and the API (content extraction API) needed to extract and verify information from those data stores. Fortunately since Oracle is a relational DBMS, it supports the standardized Java Database Connectivity (JDBC) [10] interface API, and Structured Query Language (SQL) [11] as the content extraction API.

Now our test driver for our Oracle DBMS, in order to perform its intended function, has to contain Java code that verifies the conditions in our behavioral specification (using the data from our test vectors) by extracting the security state information stored in data dictionary views through the JDBC API library calls and SQL commands. In order to generate such a test driver, we need to combine the behavioral specification and test vectors with the interface API, content-extraction API and the data dictionary views in Oracle DBMS. In other words we need information that maps the model variables in the behavioral specification to the commands in JDBC API and SQL API and the data dictionary views against which these commands must be executed. It is this mapping information that is specified in a file called 'Object Mapping File'.

With the development of the 'Object Mapping' file, the SCR behavioral specification and the test vectors we have the constituent ingredients of the verification model. The only other artifact that we need for the test driver to generate security function tests for the Oracle DBMS environment is the 'Test driver schema'.

As already stated, the test driver schemas are templates containing generic execution steps for each of the tests. In a database environment the security state is determined by a combination of security data that consists of user attributes, roles (entities that represent collection of privileges), database objects (tables, views etc) and privilege assignments to users and roles for various database objects. This security data is stored in database dictionary tables (also called system tables or database catalogues). The data in these tables cannot be created or deleted using the traditional data manipulation SQL commands but only through some privileged SQL commands. Hence definition of generic execution steps for each of the 'security function tests' against the database involves a set of these privileged SQL commands to systematically populate the database dictionary tables with security state-defining data as well as other relevant data. In other words appropriate database conditions must be established prior to the execution of each of the 'security function tests' by accessing the database as administrative-level system user.

4.4 Test Driver Code Logic and Capabilities

We now provide the taxonomy of Java programs generated by the test driver (along with their functions) in the form of a table (Table 4.3).

Table 4.3 – Generated Java Programs and their functions

Java Class Name	Function
ConfigManager	Retrieve Global Test Configuration settings (log directory, output file directory, System userid and password etc)
Constants	Provide the set of global constants used by tests
Context	Retrieve and set test vector parameters
Logger	Provides methods to write to log files and generate test output file
SQLUtils	Establish Oracle database connection through JDBC library routines
TestImpl	Specify an interface to which each test must conform along with helper methods
Test Runner	Provide a simple framework to handle the execution of the entire test

A brief description of the logic of the TestRunner class is in order at this point to provide an understanding of the logic of test execution in the generated test driver.

(1) Read global configuration file (using ConfigManager class) to determine log file directory, output file directory and the userids and passwords of users).

(2) Initialize the database under test (delete all existing security state-defining information through appropriate privileged SQL commands, define the desired security state)

(3) Get the test vectors (using TestImpl class) by called TestImpl.getTestVectors. For each test vector:
   (a) create default data (based on the data used to define the security state)
   (b) Call TestImpl.setupTest – to set up relevant data besides the security state-defining data
   (c) Call TestImpl.runTest – run the actual test and generate output (using the Logger class).
   (d) Call TestImpl.cleanupTest – restore the database to a known security state for the next test vector.
   Also perform cleanup of all other relevant data

(4) Exit

5. Conclusions

We have presented an approach (and an associated toolkit implementation) for automated security functional testing that is driven by the use of a formal behavioral model augmented with interface specifications. The development of the TAF-SFT toolkit and its application for security functional testing of a complex commercial DBMS product established the fact that both the model and the downstream test generation process are scalable. The major advantages of developing and deploying the TAF-SFT toolkit for security functional testing are:

(a) Better quality of specifications and quality of test data
(b) Automated generation of executable test code and automated results analysis.

The major disadvantages are the detailed knowledge of the security function semantics required on the part of the modeler to develop good behavioral models and the complexity of object mapping information that may result in case of products with complex interfaces. These disadvantages can be partially overcome in situations where the following are possible:

(a) Partial reuse of SCR security behavioral model
(b) Partial reuse of Object Mapping information

Since the SCR behavioral model is based on the security function specifications, reuse of parts of this model is possible if security function specifications in the different products under security testing are based on an interoperable security API like CDSA [12]. Partial reuse of Object Mapping information is possible if the different products under security testing support a common interface API (like the different relational DBMS products supporting the JDBC API and the SQL API).

6. References

[1] E.L.Safford. "Key applications of Test Automation Framework (TAF)", Proc 12th Annual Software Technology Conference, April 30 - May 5, 2000.

[2] D.Statezni. "Test Automation Framework, State-based and Signal Flow Examples", Proc. 12th Annual Software Technology Conference, April 30 – May 5, 2000.

[3] W.Jansen. "Security Testing Characteristics", http://csrc.ncsl.nist.gov/sectest/Security_Testing.html, April 1998

[4] C.Heitmeyer, J.Kirby,B.Labaw and R.Bharadwaj. "SCR: A toolset for specifying and analyzing software requirements",Proc. 10th Annual Conference on Computer-Aided Verification, Vancouver, Canada, 1998.

[5] M.R.Blackburn, R.D. Busser, "T-VEC: A Tool for Developing Critical System", Proc. 11th International Conference on Computer Assurance, Gaithersburg, Maryland,USA pages 237-249, June, 1996.

[6] M.R.Blackburn., R.D. Busser, J.S. Fontaine. "Automatic Generation of Test Vectors for SCR-Style Specifications", Proc. 12th h Annual Conference on Computer Assurance, Gaithersburg, Maryland, pages 54-67, June, 1997.

[7] S.R.Faulk, P.C.Clements. "The NRL SCR requirements specification", Proc. 4th International Workshop on Software Specification and Design, Monterey, California,USA, 1987.

[8] Oracle Corporation, Oracle8 Security Target Release 8.0.5, April 2000.

[9] ISO/IEC International Standard (IS) 15408, http://csrc.nist.gov/cc/ccv20/ccv2list.htm

[10] JDBC Data Access API, http://java.sun.com/products/jdbc/download.html

[11] ISO/IEC 9075:1999, "Information Technology --- Database Languages --- SQL", http://www.iso.org

[12] Common Data Security Architecture (CDSA), http://www.opengroup.org/security/l2-cdsa.htm