applied sciences

MDPI

Article

Maximizing Test Coverage for Security Threats Using Optimal Test Data Generation

Talha Hussain [1], Rizwan Bin Faiz [1], Mohammad Aljaidi [2,*], Adnan Khattak [1], Ghassan Samara [2], Ayoub Alsarhan [3] and Raed Alazaidah [2]

[1] Faculty of Computing, Riphah International University, Islamabad 46000, Pakistan; thussain98ml@gmail.com (T.H.); rizwan.faiz@riphah.edu.pk (R.B.F.); adnan_ktk08@yahoo.com (A.K.)
[2] Department of Computer Science, Faculty of Information Technology, Zarqa University, Zarqa 13110, Jordan; gsamara@zu.edu.jo (G.S.); razaidah@zu.edu.jo (R.A.)
[3] Department of Information Technology, Faculty of Prince Al-Hussein Bin Abdallah II for Information Technology, The Hashemite University, Zarqa 13116, Jordan; ayoubm@hu.edu.jo
* Correspondence: mjaidi@zu.edu.jo

Abstract: As time continues to advance, the need for robust security threat mitigation has become increasingly vital in software. It is a constant struggle to maximize test coverage through optimal data generation. We conducted explanatory research to maximize test coverage of security requirements as modeled in the structured misuse case description (SMCD). The acceptance test case is designed through the structured misuse case description for mitigation of security threats. Mal activity is designed from SMCD upon which constraints are specified in object constraint language (OCL) in order to minimize human dependency and improve consistency in the optimal test case design. The study compared two state-of-the-art test coverage maximization approaches through optimal test data generation. It was evident through the results that MC/DC generated optimal test data, i.e., n + 1 test conditions in comparison to the decision coverage approach, i.e., 2^n test conditions for security threats. Thus, MC/DC resulted in a significantly lower number of test cases yet maximized test coverage of security threats. We, therefore, conclude that MC/DC maximizes test coverage through optimal test data in comparison to decision coverage at the design level for security threat mitigation.

Keywords: modified condition/decision coverage; decision coverage; test coverage; test data; object constraint language; structured misuse case description; system under test

Citation: Hussain, T.; Faiz, R.B.; Aljaidi, M.; Khattak, A.; Samara, G.; Alsarhan, A.; Alazaidah, R. Maximizing Test Coverage for Security Threats Using Optimal Test Data Generation. *Appl. Sci.* **2023**, *13*, 8252. https://doi.org/10.3390/app13148252

Academic Editors: Paolino Di Felice and Vito Conforti

Received: 29 April 2023
Revised: 28 May 2023
Accepted: 16 June 2023
Published: 16 July 2023

1. Introduction

Software testing is a time-consuming but vital activity that tries to raise the quality of software [1]. The prime goal of software testing is to ensure that the delivered product is bug-free. However, due to the constraints of the testing project's budget, manpower, and testing time, it would be unrealistic to expect perfect and faultless software to successfully complete the project. Therefore, it is essential to manage the testing cost to reduce the overall development budget and time [2]. As a result, rather than spending a large budget pursuing a flawless software system, most software developers would make a compromised testing plan [3]. Test coverage is an important indicator of software testing [4]. Offering data on various coverage item aids in evaluating the success of the testing process. The hardest part of maximizing test coverage is coming up with appropriate test data [5]. Moreover, generating test data to achieve 100% coverage is labor-intensive and expensive. A greater number of tests also necessitates a longer test period and greater tester memory. Test development becomes more challenging due to the growing number of tests required to fulfil adequate coverage criteria [6]. Selecting a portion of tests from a large baseline test set becomes critical when it is impossible to apply all the tests due to test time or tester memory limitations. It is clear from the

literature that testing accounts for more than 40% of project costs [7]. Instead of covering all the entities in the tested program, test data can be aimed toward covering the optimal set to reduce the testing effort [8]. Reducing the amount of data utilized for testing is one method of minimizing this cost. Test data optimization can maximize test coverage, yet reduce maintenance cost, for both black-box and white-box testing.

Our research goal is to maximize test coverage through optimal test data generation at the design level for security threat mitigation. Early security threat detection and corresponding mitigation leads to a reduction in maintenance costs, especially in the case of large software systems [9]. Time and cost can be saved if the security defects are caught in the early stage of the development, but it is most costly if the defects are found later. It also causes damage to the organizations financially or in terms of data loss or user inconvenience. Following on the motivational work of early testing [10], we have used the mal-activity diagram for our experiment with constraints specified in object constraint language (OCL). UML models with specified constraints in OCL are precise, comprehensive, and consistent [11]. These constraints were derived from basic and alternate flow given in the structured misuse case description and were modeled in the mal-activity diagram. However, to complete the UML model, constraints can only be specified in formal constraint language, i.e., object constraint language (OCL).

The modified condition/decision coverage (MC/DC) methodology is one widely used method for generating the optimum test data for code coverage [12]. Through optimal test data generation, we aim to maximize test coverage in our study. Achieving 100% test coverage can be thought of as maximizing test coverage, since it is noted in the literature that the method used to generate test data to reach 100% coverage is laborious and expensive. This study concludes that achieving 100% test coverage is not realistic or feasible.

Use cases outline functional specifications; however, they are unable to allow the modeling of security threats. To model security threats and identify misuse cases, the use case diagram is developed. Since a use case model simply specifies the necessary capabilities, the textual description often captures the core of the use case. This textual description plays a vital role while representing a misuse case. Diagrams of misuse cases and the textual descriptions of those diagrams give developers essential security-related details.

Misuse cases can be used to implement security threats, and test cases are developed from misuse cases for security threats. Diagrams of misuse cases and the textual descriptions of those diagrams give the developer essential security-related details [13]. Techniques to create security test cases from misuse instances have been proposed by a plethora of authors [14]. By executing test cases and verifying that the software worked as intended, security testing may be put into practice utilizing misuse cases. An example of a test case would include test input, excepted output, and actual output. It is a sign that the program functionality is correctly implemented when the expected output and actual output match.

We create test cases using (i) automated and brute, (ii) weak password attacks, (iii) session id links, and (iv) session expiry time. The goal of employing misuse situations is to build security acceptance test cases at a high level and get beyond the challenge of discovering acceptance test cases from programming languages. Without any knowledge of the programming language, the user will concentrate on creating acceptance test cases from textual descriptions. To create acceptance test cases, textual descriptions of the use cases, misuse instances, threats to them, and solutions are identified. The various usage scenarios should be covered by the designed acceptance test cases. Security acceptance test cases include expected outputs and inputs like any other test case. Data or functional calls might be used in the input security acceptance test scenarios. Evaluation of acceptance test cases is done in the output of the intended outcome.

The remainder of this paper is structured as follows: The Literature Review, Research Question and Gap Analysis are discussed in Section 2. The Research Methodology is discussed in Section 3. The Experiment design and complete methodology are discussed in Section 4. The acceptance test cases created for security risks from both SMCD and USMCD

are also consistently evaluated in Section 4 of the report. Results and threats to validity are discussed in Section 5. The conclusion and future work are presented in the last Section 6.

Research Significance

Our study's research significance is:

1. Firstly, it will help in designing consistent acceptance test cases for security threats (authentication and authorization) through structured misuse case descriptions for early-stage mitigation of security threats. This will help us to overcome the challenge of inconsistent acceptance test case design due to its reliance upon human judgment.
2. Secondly, by comparing two state-of-the-art approaches that maximize test coverage through optimal test data generation for structured misuse case descriptions. It was evident from our results that modified condition decision coverage (MC/DC) maximizes test coverage through optimal test data generation with minimum test conditions in contrast to decision coverage (D/C) through structured misuse case descriptions.

2. Literature Review

Test coverage is a crucial component of software maintenance and a significant indicator of the quality of the product. By offering information on various coverage topics, it aids in evaluating the success of the testing process. It takes a lot of time and money to create enough test data to reach 100% coverage. A greater number of tests necessitates a longer testing period and more tester memory. However, the quantity of tests needed to guarantee high coverage criteria has been rising, making test development more difficult [15]. A subset of tests from a large baseline test set must be chosen when it is difficult to apply all the (alleged) tests due to test time or tester memory limitations. Test data generation can be aimed towards covering the ideal set rather than all the entities in the tested program to reduce the testing effort. We generate a large amount of test data when we conduct model-based testing. The main challenge in software testing is coming up with test data that meets a specific adequacy criterion. As we will have different test data at the design level, more precisely when we conduct the testing manually, this problem results from the test case subjectivity issue. The test case subjectivity issue will be fixed because the design diagram was made precise with constraints specified in object constraint language (OCL) [11]. Techniques like MC/DC and decision coverage are used at the code level to increase test coverage. In [16] applied MC/DC at graphics processing units (GPUs) and sequential (central processing unit) CPU code and proved how you can achieve 100% MC/DC code coverage in combinational testing. However, to the best of our knowledge, it has not yet been determined in the literature review which of these two, (MC/DC) or (D/C), maximizes test coverage through ideal test data generation at the design level, so we'll use both of these strategies and compare the outcomes.

Test coverage maximization has been exercised in a number of ways in the literature. Subhash and Vudatha maximized test coverage through combinatorial test cases using the particle swarm optimization algorithm [17]. Avdeenko increased code coverage through automated test data generation based on a genetic algorithm [18]. In other research by Lemieux [19], test coverage was also improved through search-based software testing (SBST), which generates high-coverage test cases with a combination of test case generation and mutation at the code level.

Heba Muhammad and Muhammad Najm in [20] maximized test coverage through optimal test data generation by using the hybrid greedy hill climbing algorithm (HGHC) for generating a small number of test data. Gupta maximized test coverage through the test suite minimization approach based on diversity aware mutation adequacy criterion for detecting and locating faults together [21]. Shahbaa increased test coverage through test case prioritization techniques [22]. A recent study published by Barisal and Kumar was related to developing methods for maximizing the structural code coverage for MC/DC [23].

Thus, there is strong research motivation to apply MC/DC at the design level so as to maximize test coverage for security threats.

In [16], Jaime Luis maximized test coverage at the code level through MC/DC. Besides in [24], the author used MC/DC to maximize test coverage for software under test (SUT) through optimal test data generation for a search-based empirical evaluation. Noman [25] maximized test coverage at the source code level through MC/DC in combination with manual testing.

A UML diagram can be used as input for a test case generation technique, and test cases can then be validated using another technique [17]. Moreover, when using UML models we need to ensure that the generated code is correct for the UML Model. This requires the UML models to be accurate, complete, and consistent. Any errors or inconsistencies in the UML models can lead to incorrect code generation [26]. The literature makes a strong case for the value of constraints specified in OCL at the design level. By outlining models that cannot be included, constraints specified in OCL help UML models be comprehensive, consistent, and accurate when used with them [11]. The unified modeling language (UML), which receives a lot of interest from scholars and professionals, has become a crucial standard for modeling software systems [27]. Extensive test data is produced when we conduct model-based testing [28]. Making test data that meet a specified adequacy criterion is a key challenge in software testing.

In [29], the author has generated acceptance test cases for security threats using an unstructured misuse case description. By incorporating the misuse case description into the mal-activity diagram, a misuse scenario is created and inputs and triggers are determined. The flow of usage shown in the mal-activity diagram is used to design test cases. Test cases are created in [30] where misuse instances pose security issues. The findings show that the suggested misuse case creation method offers superior coverage for security issues. Their strategy, nevertheless, has to be better organized and provide a thorough explanation of the procedure. The techniques used to generate test cases from the use case were extensively available in the latest research. A use case model is initially created from the functional requirements. As a result of the use case model, test cases are created. The final stage involves using commercially available tools to execute the prepared test cases. In [31], the author proposed an approach to generate test cases from use cases, misuse cases, and 137 mitigations of use case descriptions. Early on in a product's development, this includes security features. In misuse scenarios, they recommend several improvements to make it easier to define security needs.

It is evident from the above literature that there is a need to design consistent acceptance test cases for security threats (authentication and authorization) through structured misuse case descriptions for early-stage mitigation of security threats. This will help us to overcome the challenge of inconsistent acceptance test case design due to its reliance upon human judgment. For that, we have first identified misusage scenarios through structured misuse case descriptions against security threats such as authentication and authorization and their corresponding mitigation. We then model a combination of basic and alternate security threat mitigation scenarios in the mal-activity diagram.

Besides two state-of-the-art approaches that maximize test coverage through optimal test data generation from source code [16,32]. However, which among the modified condition decision coverage (MC/DC) and decision coverage (D/C) maximizes test coverage through optimal test data generation with minimum test conditions at the design level is yet to be explored. Besides, a combination of MC/DC with manual testing at the design level has not yet been experimented on.

Research Question

The main challenge in software testing is coming up with test data that meets an established adequacy criterion. It is evident from the literature, that test coverage in source code can be maximized through optimal test data generation using the MC/DC and decision coverage approach. However, there is no evidence in the literature on how to

maximize test coverage for SMCD through optimal test data. To achieve our goal, we focus on the following research questions:

- RQ1: Which among decision coverage and MC/DC maximizes test coverage for security threats in the Structured Misuse case description?
- Hypothesis 1: Decision coverage maximizes test coverage for security threats in the Structured Misuse case description
- Hypothesis 2: MC/DC maximizes test coverage for security threats in the Structured Misuse case description.

3. Research Methodology

This section defines the thorough facts of our conducted experiment. Let us discuss the content in detail. Our research approach is quantitative because we have a numerical dataset and we will use statistical analysis methods to test relationships between variables. We apply MC/DC and decision coverage criteria on the Design Level for optimal test coverage of security requirements generating the Optimal Test and improving consistency for security requirements. We will perform an experiment to generate optimal test coverage to reduce the testing effort by achieving maximum test coverage through optimal test data.

Our research methodology is explanatory research, which aims to explain the causes and consequences of a well-defined problem. Spending too much time in testing is a well-defined problem. There are often independent and dependent variables in a control experiment. Our research uses the experimental method to determine the cause and effect between variables, including dependent and independent variables. In our experiment, the dependent variable is the Optimal Set of test data, and the independent variables are MC/DC and decision coverage.

4. Experiment Design

This study aims to maximize test coverage for security threats using Optimal Test Data Generation. One of the most common ways to model security threats is through the misuse case. To achieve our research goal, we undertake an experiment to generate optimal test data at a design level. In the experiment, test data was generated through (MC/DC) and decision coverage from the mal-activity diagram, with test conditions extracted for seven constraints. Overall, this study contributes to the field of software security by providing a methodology for generating optimal test data at the design level for security threats. Moreover, test data generated from OCL constraints eliminated the challenge of inconsistent acceptance test case design due to its reliance upon human judgment.

The steps of the proposed methodology are as follows:

1. Identify security authentication and authorization threats.
2. Design the structured misuse case description.
3. Draw mal-activity from the structured misuse case description.
4. Specify constraints in the mal-activity diagram using OCL.
5. Transform constraints into Boolean expression.
6. Transform Boolean expression into Truth Table Expression.
7. Generate possible test data:
   - 7.1 through MC/DC (modified condition decision coverage).
   - 7.2 through decision coverage (D/C).
8. Compare and find optimal test data generated through MC/DC and D/C
9. Design test cases for generated optimal test data.

In Figure 1 as mentioned below, above steps are mapped according to the methodology flow.

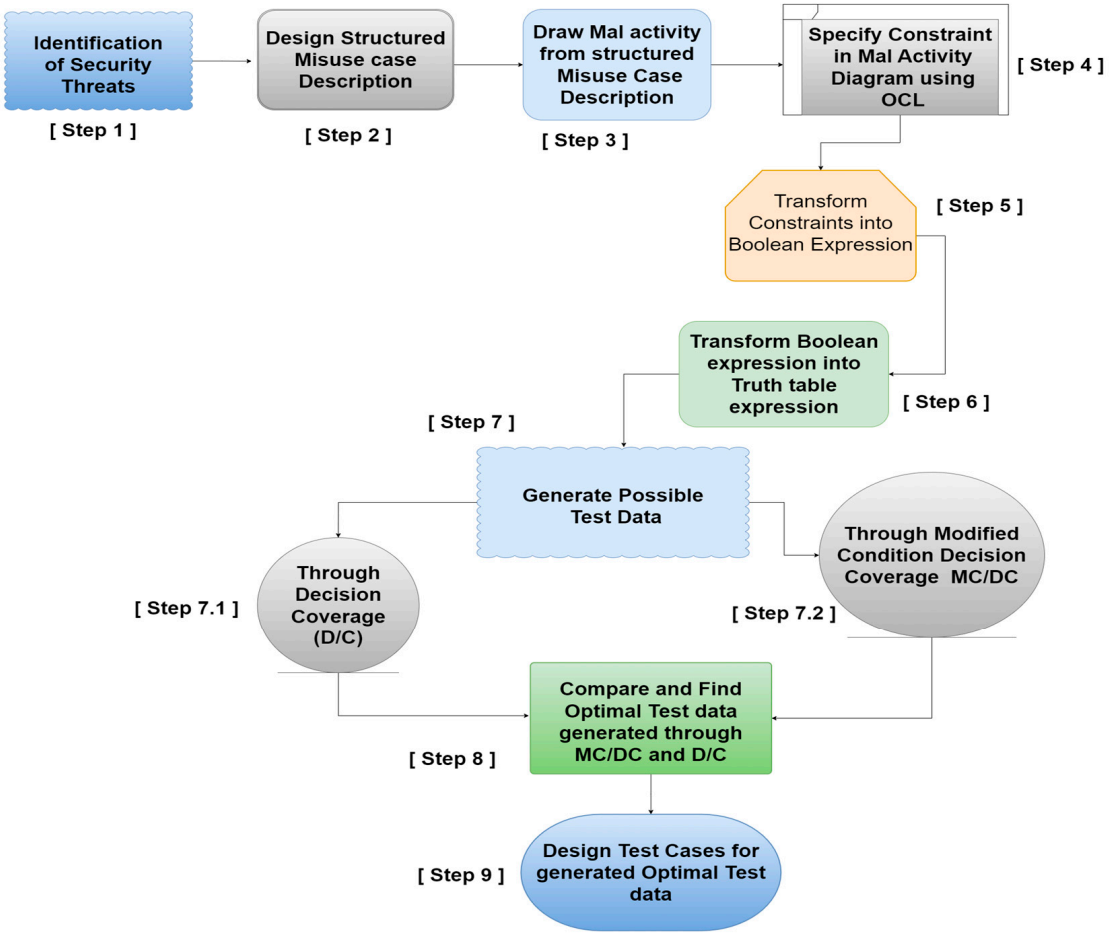

Figure 1. Proposed Methodology.

Step 1: Elicitation/Extraction of Security Requirements i.e., authentication and authorization

In the context of software development, security requirements refer to the set of features and functionalities that ensure that the software system is secure and protected against unauthorized access, data breaches, and other types of cyber-attacks. The first step in developing a secure software system is to elicit or extract the security requirements.

"Elicitation/Extraction of Security Requirements" refers to the process of identifying and defining these security requirements. In Table 1, Authentication and Authorization were addressed.

Table 1. Identification of Security Threats.

Elicitation of Security Requirement	Goal	Sub-Goal
The application should authenticate the User using a valid username and Password.	Security	Authentication
Authorization codes should be set up and modified only by the System Administrator.	Security	Authorization

Step 2: Structured Misuse Case Description

The structured misuse case description is provided in the following Table 2.

Table 2. Structured Misuse Case Description.

ID*	SMC-SA-001
Goal*	Security
Sub Goal*	Authentication
Misuse Case Name*	Steal Login Details IMPLEMENTS steal sensitive data
Associated Misusers*	Information Thief
Author Name	ABC
Date	dd/mm/yy
Description*	Misuser gets access through automated attacks such as credential stuffing and brute force technique to Login into the system to perform illegal activities with the user data.
Preconditions*	The login page is accessible to the Information Thief.
Trigger*	Information Thief clicks the login button
Basic Flow*	Information Thief uses an automated attack tool to generate many combinations of usernames and passwords. Login Details, i.e., username, Password, and Captcha matched with the login details of the system. On Successful Login, a verification code will be sent on the user email id/SMS for multifactor authentication. If a user receives a verification code and verifies the login attempt, then the Information thief will be redirected to the User's personal and sensitive data pages.
Alternate Flow*	**BF-2**. In case of non-authentic/invalid login details, i.e., username, Password, and Captcha, or the number of login attempts are greater than three against the same IP, it will be blocked. **BF-4**. If the multifactor authentication verification code is not received through Email/SMS, repeat the Bf3.
Assumption	The system has login forms feeding input into database queries.
Threatens Use Case*	User Login
Business Rules	The Hospital system shall be available to its end-users over the internet.
Stakeholder & Threats	Hospital O/I Maintenance Department, O/I User Department, Store Keeper, Dispenser. If deleted, data loss reveals sensitive information to damage the business and reputation of the hospital.
Threatens Use case Mitigation	If a user from the same IP address attempt three logins failed attempts, block the IP. Also, apply Captcha and multifactor authentication to avoid attacks.
Note:	All fields with * are mandatory for Structured Misuse Case Description.

Step 3: Designing the Mal-Activity Diagram

In this step, the mal-activity diagram is designed for the Structured Misuse case description. Structured Misuse case descriptions are modeled into the mal-activity diagram to identify the inputs and triggers. Figure 2 illustrates the mal-activity diagram modeled from the structured misuse description.

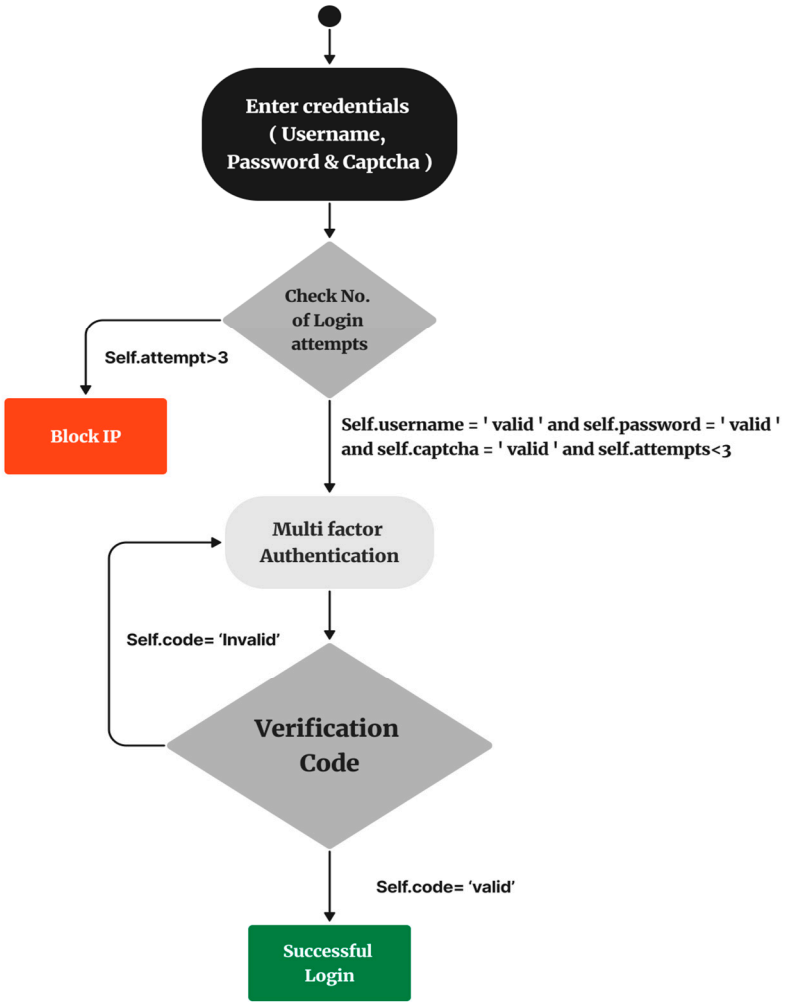

Figure 2. Mal-activity diagram for SMC-SA-001.

Step 4: Specify constraints in Mal Activity Diagram using OCL

Following are the specification of constraints in OCL that will be used for transformation into boolean expression.

self. Username = 'valid' and self. Password = 'valid' and self.captcha = 'valid'} and self.attempts < 3

Step 5: Transformation of Constraints into Boolean Expression

self. Username = 'valid' ∧ self. Password = 'valid') ∧ (self.captcha = 'valid' ∧ self.attempts < 3))

After converting into a logical operation, we applied the MC/DC and decision coverage to each constraint.

Step 6: Transform Boolean expression into truth table expression

We will specify and label each constraint separately, i.e., Table 3, before transforming them into Boolean expression.

Table 3. Explanation of constraints transformation.

a	self. Username = 'valid'
b	self. Password = 'valid'
c	self.captcha = 'valid'
d	self.attempts < 3

Constraints specified using Table 3 become ((a ∧ b) ∧ (c ∧ d)).

In the following Table 4, the Boolean expression, which is first transformed from a constraint, is now transformed into truth Table form.

Table 4. Transformation of Boolean Expression into truth table form.

No.	a	b	c	d	((a ∧ b) ∧ (c ∧ d))
1	F	F	F	F	**F**
2	F	F	F	T	**F**
3	F	F	T	F	**F**
4	F	F	T	T	**F**
5	F	T	F	F	**F**
6	F	T	F	T	**F**
7	F	T	T	F	**F**
8	F	T	T	T	**F**
9	T	F	F	F	**F**
10	T	F	F	T	**F**
11	T	F	T	F	**F**
12	T	F	T	T	**F**
13	T	T	F	F	**F**
14	T	T	F	T	**F**
15	T	T	T	F	**F**
16	T	T	T	T	**T**

Step 7: Generating Possible Test

We will use the decision coverage approach mentioned in [32,33] and MC/DC described in [16]. To obtain MC/DC and decision coverage, we need to solve the combinations of true and false values required to achieve the MC/DC criterion. Multiple solutions are required corresponding to a constraint to generate test data according to the MC/DC criterion. For example, consider a constraint as an expression C = p ∨ q, where p and q are the clauses of the constraint. There are four combinations of possible outcomes, two each for p and q (TT, TF, FF, and FT).

Step 7.1: Generating test data for MC/DC

The idea of the MC/DC is to select the subset of all possible combinations that directly impact the outcome value of the actual constraint. In the case of C, these combinations are (FF, TF, and FT). To identify this subset, the first step is to reformulate the original constraint to obtain more constraints that satisfy the MC/DC criterion.

The pair table suggested in [34] provides several potential pairs for each clause, and we need to select minimum subsets of pairs that cover all clauses. In our table, potential pairs are (8,16) from A, (12,16) from B, (14,16) from C, and (15,26) from D. For A, we need to test only conditions of (8,16); for B, we need to test only (12,16). For C, we need to test only (14,16); for D, we must only test (15,16). So, we need to test 5 constraints (8, 12, 14, 15, 16) as specified in Table 5.

Table 5. Color Mapping for potential Constraints found in MC/DC.

No.	a	b	c	d	((a ∧ b) ∧ (c ∧ d))
1	F	F	F	F	**F**
2	F	F	F	T	**F**
3	F	F	T	F	**F**
4	F	F	T	T	**F**
5	F	T	F	F	**F**
6	F	T	F	T	**F**
7	F	T	T	F	**F**
8	F	T	T	T	**F**
9	T	F	F	F	**F**
10	T	F	F	T	**F**
11	T	F	T	F	**F**
12	T	F	T	T	**F**
13	T	T	F	F	**F**
14	T	T	F	T	**F**
15	T	T	T	F	**F**
16	T	T	T	T	**T**

Step 7.2: Generating test data for D/C

Decision coverage requires test cases to cover both branches of a decision. For each decision, the D/C criterion requires two test cases. For example, for a decision, the Boolean expression requires two test cases, e.g., the test cases (A = true, A = False) [33].

We have 16 test combinations for decision coverage because we must have to test whether each combination is false or true for each condition, as mentioned in [33].

Step 8: Compare and Find Optimal Test Data

In the above example, we have 16 test conditions; for decision coverage, we need to test all 16 test conditions to achieve maximum coverage for MC/DC. Therefore, it is required to test only 5 test conditions, which comprise the actual test conditions in terms of finding maximum errors. For MC/DC, optimal test conditions were found, and for decision coverage, it is required to test all conditions. So our hypothesis 2 'MC/DC maximizes test coverage for security threats in the Structured Misuse case description' is true and hypothesis 1 will be negated in the results of this experiment.

In step 8, we find that test data generated for MC/DC requires fewer test combinations as compared to decision coverage. Therefore, in the test case design, we will use test combinations identified for MC/DC.

4.1. Test Combinations Identified by MC/DC:

There were total 16 constraints generated truth table as mentioned in Table 4. After applying MC/DC found test conditions in MC/DC are 8, 12, 14, 15, 16 as specified in Table 5. These constraints are identified after application of MC/DC on relevant OCL value for A,B,C,D. In above Table 5, color mapping is given for relevant OCL value i.e for A,B,C,D. For **a** potential constraints are 8,16 mapped in orange color. Similarly, for **b** potential constraints are 12,16 mapped in grey color, for **c** 14,16 mapped in green and for **d** for 15,16 in purple. In result we found that the test combinations are 8,12,14,15,16 as specified in Table 5, for which the test case will be designed.

Step 9: Acceptance test case design through Structured Misuse Case Description

In step 8, we identified that potential constraints are 8, 12, 14, 15 and 16. In order to design test case from these constraints we need to check the truth/false value for each a,b,c,d truth table value. In case if the value is true constraint will remain same, incase if it's false it will be reversed e.g., in given 8th constraint, we have F,T,T,T values for **a,b,c**

and **d** respectively. We need to modify the constraint value for **a** since we got F result for potential 8th constraint. In original Constrain we have a = self. Username = 'valid' referred to Table 1, after altering the F value it will become self. Username = 'Invalid' and rest of the constraint will remain same. For all these potential constraints we have updated the original constraint accordingly in following Table 6.

Table 6. Identified Test Combination for MC/DC.

Constraint No.	Test Scenario	Constraint
8	Invalid Username Unsuccessful login	self. Username = 'Invalid' ∧ self. Password = 'valid') ∧ (self.captcha = 'valid' ∧ self.attempts < 3)
12	Unsuccessful Login due to invalid Password	self. Username = 'valid' ∧ self. Password = 'Invalid') ∧ (self.captcha = 'valid' ∧ self.attempts < 3))
14	Unsuccessful Login due to invalid Captcha	self. Username = 'valid' ∧ self. Password = 'valid') ∧ (self.captcha = 'Invalid' ∧ self.attempts < 3))
15	Unsuccessful Login due to more than three login attempts	self. Username = 'valid' ∧ self. Password = 'valid') ∧ (self.captcha = 'valid' ∧ self.attempts > 3))
16	Successful Login	self. Username = 'valid' ∧ self. Password = 'valid') ∧ (self.captcha = 'valid' ∧ self.attempts < 3)))

4.2. Acceptance Test Case Design

Acceptance test cases through the structured misuse case description (SMCD) will be designed from the above mal-activity diagram shown in Figure 1. Test data is essential as it is used to execute test cases [35]. We use the equivalence class partitioning technique to generate test data as referred to in [36]. We have two scenarios in the above activity diagram, and test cases will be designed for both scenarios. In the first scenario, users enter their username and password to login. The IP of the system will be blocked for more than three invalid attempts from the same IP. When the login and password are valid and the user has made fewer than three attempts, the system will create a verification code. The user will enter the verification code that was delivered to the email address in the second scenario. If the code entered is authentic, the system will log the user in; otherwise, they must repeat scenario 1 until they obtain the code. The acceptance test scenarios for incorrect usernames, passwords, Captchas, and login attempts are shown.

For this situation, we will now create a test case. Username Valid Class: {A–Z}, {a–z} and Invalid Class: {0–9}, {!@#$%^&*()[]{}}, Password Valid Class: {A–Z},{a–z}, {0–9},{!@#$%^&*();:[]{}} and Invalid Class: {A–Z},{a–z} Captcha Valid Class: {A–Z}, {a–z},{0–9} and Invalid class: {!@#$%^&*();:[]{}} Login Attempts Valid Class:{0 < attempt $\leq$ 3} and Invalid Class: {attempt > 3}.

In the following Table 7, an acceptance test case is designed for the structured Misuse case description.

The designed test cases for the identified test combinations have been effective in detecting errors, while no other test combinations were found to be as effective other than those mentioned in MC/DC. This result has given us satisfaction with our approach and answered our question that MC/DC has generated test data with optimal test conditions and it was also proved after the test case design. Results from the experiment and test case design concludes that the test data generated for MC/DC is optimal and consistent at the design level as proved by the formal activity diagram.

Table 7. Acceptance test case Design.

TC #	Scenario	ECP	Input				Expected Output
			Username	Password	Captcha	Attempts	
TC-08	Invalid Username Unsuccessful login	Username: Invalid Class: {0–9, @#$%^&*()[]{}}	Admin123	Admin@98ml	As12	1	Unsuccessful Login due to the wrong username
		Password: Valid: {A–Z, a–z, 0–9, !@#$%^&*();:[]{}}		admin_#@!11			
		Captcha: Valid Class: {A–Z},{a–z},{0–9}			AB23C		
		Login Attempts: Valid: {0 < attempt $\leq$ 3}				2	
TC-SA-12	Unsuccessful Login due to invalid Password	Username: Valid: {A–Z, a–z}	user				Unsuccessful Login due to the wrong Password
		Password:Invalid Class: Password = {A–Z},{a–z}		1234			
		Captcha: Valid Class: {A–Z},{a–z},{0–9}			XYZ88		
		Login Attempts: Valid: {0 < attempt $\leq$ 3}				3	
TC-SA-14	Unsuccessful Login due to invalid Captcha	Username: Valid: {A–Z, a–z}	ABC				Unsuccessful Login due to invalid Captcha
		Passsword: Valid: {A–Z, a–z, 0–9, !@#$%^&*();:[]{}}		Admin&12345			
		Captcha: Invalid class: {!@#$%^&*();:[]{}}			ZX&12		
		Login Attempts: Valid Class: {attempt > 3}				1	
TC-SA-15	Unsuccessful Login due to more than three login attempts	Username: Valid: {A–Z, a–z}	User				Unsuccessful Login due to more than three login attempts
		Password: Valid: {A–Z, a–z, 0–9, !@#$%^&*();:[]{}}		Admin&123			
		Captcha: Invalid class: {!@#$%^&*();:[]{}}			ZXC12		
		Login Attempts: Invalid Class: {attempt > 3}				4	
TC-SA-16	Successful Login	Username: Valid: {A–Z, a–z}	User				Successfully logged in
		Passsword: Valid: {A–Z, a–z, 0–9, !@#$%^&*();:[]{}}		Admin&123			
		Captcha: Valid Class: {A–Z},{a–z},{0–9}			ZXC12		
		Login Attempts: Valid Class: {0 < attempt $\leq$ 3}				1	

5. Results and Discussion

In this research, we have constraints with four n values. Constraints specified in OCL can be with different n values. The test combinations identified are 5 for MC/DC and 16 for D/C. It is evident from the results that the test data generated for MC/DC is optimal in each design experiment used. We performed a succinct analysis of the findings for n = 1,

n = 2, n = 3, and n = 5. Additionally, we thoroughly specified the complete test data generation procedure for n = 4 in the experiment given in the paper. It is important to note that for experiments involving n = 2, n = 3, and n = 5, test data has been generated. Furthermore, it should be noted that no constraint was accessible for n > 5. Based on the conclusions drawn from the findings of this study, a mathematical formula has been formulated for MC/DC and D/C approaches. The details of this formula are elaborated upon in the concluding section of this paper.

The following formula is derived for MC/DC based on test data generated for all design diagrams where n = 2, 3, 4, 5, respectively, for each diagram.

$$\sum t = n + 1 \tag{1}$$

The following formula is derived for D/C on basis of the test data generated for all design diagrams where n = 2, 3, 4, 5, respectively, for each diagram.

$$\sum t = 2^n \tag{2}$$

The test condition results clearly show MC/DC has generated optimal test data from constraints compared to decision coverage. As a next step, we have designed the test cases for optimal test conditions through equivalence class partitioning. The resultant scenarios after test case generation proved that no other test scenario is identified except those which were already extracted from MC/DC test data.

Threats to Validity

The groups chosen for experimentation might differ before receiving any treatment. Because we have only utilized MC/DC and decision coverage to produce test data at the design level, it could be dangerous if suddenly decision coverage performed better when expressions became more complex or UML diagrams were altered. Additionally, if results are altered for higher constraint orders, i.e., more than five, the formula obtained from the extracted data results may also alter.

6. Conclusions and Future Work

This research designs acceptance test cases for security threats through SMCD. In order to maximize test coverage through optimal test data generation, we model security threat mitigation in SMCD. A mal-activity diagram was designed from SMCD to generate consistent test data through MC/DC and D/C. A comparative analysis explains that *MC/DC* maximizes test coverage through optimal test data from SMCD in comparison with decision coverage. We have generated the test data from constraints for both MC/DC and decision coverage. MC/DC generated n + 1 test conditions unlike decision coverage, which generated test conditions. Moreover, all the test data generated from the constraint is the same in each case, irrespective of the tester. Therefore, consistent test data will be generated through MC/DC or decision coverage due to the constraint specification in the object constraint language, hus, reducing the test case subjectivity. In this experiment, regular expression orders up to order 4 are employed. In the future, it is possible to conduct experiments for constraint n > 5 with higher complexity for other UML diagrams, and other test data generation approaches can use MC/DC and D/C at the design level for extensive analysis.

Author Contributions: Conceptualization, T.H. and R.B.F.; data curation, T.H. and R.B.F.; formal analysis, T.H. and R.B.F.; funding acquisition, M.A., G.S. and R.A.; investigation, T.H., R.B.F., A.K. and M.A.; methodology, T.H.; project administration, R.B.F., M.A., A.A. and G.S.; resources, T.H.; software, T.H.; supervision, R.B.F.; validation, T.H., R.B.F., A.K., M.A. and A.A.; writing—original draft, T.H.; writing—review and editing, T.H., R.B.F., R.A. and G.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research is funded by Zarqa University.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The authors would like to extend their sincere appreciation to Zarqa University for supporting this research.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bharathi, M. Hybrid Particle Swarm and Ranked Firefly Metaheuristic Optimization-Based Software Test Case Minimization. *Int. J. Appl. Metaheuristic Comput.* **2022**, *13*, 1–20. [CrossRef]
2. Habib, A.S.; Khan, S.U.R.; Felix, E.A. A systematic review on search-based test suite reduction: State-of-the-art, taxonomy, and future directions. *IET Softw.* **2023**, *17*, 93–136. [CrossRef]
3. Huang, T.; Fang, C.C. Optimization of Software Test Scheduling under Development of Modular Software Systems. *Symmetry* **2023**, *15*, 195. [CrossRef]
4. Aghababaeyan, Z.; Abdellatif, M.; Briand, L.; Ramesh, S.; Bagherzadeh, M. Black-Box Testing of Deep Neural Networks Through Test Case Diversity. *IEEE Trans. Softw. Eng.* **2023**, *49*, 3182–3204. [CrossRef]
5. Mohi-Aldeen, S.M.; Mohamad, R.; Deris, S. Optimal path test data generation based on hybrid negative selection algorithm and genetic algorithm. *PLoS ONE* **2020**, *15*, e0242812. [CrossRef]
6. Wang, J.; Lutellier, T.; Qian, S.; Pham, H.V.; Tan, L. EAGLE: Creating Equivalent Graphs to Test Deep Learning Libraries. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 22–27 May 2022; pp. 798–810. [CrossRef]
7. Khari, M.; Sinha, A.; Verdú, E.; Crespo, R.G. Performance analysis of six meta-heuristic algorithms over automated test suite generation for path coverage-based optimization. *Soft Comput.* **2020**, *24*, 9143–9160. [CrossRef]
8. Alomar, E.A.; Wang, T.; Raut, V.; Mkaouer, M.W.; Newman, C.; Ouni, A. Refactoring for reuse: An empirical study. *Innov. Syst. Softw. Eng.* **2022**, *18*, 105–135. [CrossRef]
9. Sidhu, B.K.; Singh, K.; Sharma, N. A machine learning approach to software model refactoring. *Int. J. Comput. Appl.* **2022**, *44*, 166–177. [CrossRef]
10. Pachouly, J.; Ahirrao, S.; Kotecha, K.; Selvachandran, G.; Abraham, A. A systematic literature review on software defect prediction using artificial intelligence: Datasets, Data Validation Methods, Approaches, and Tools. *Eng. Appl. Artif. Intell.* **2022**, *111*, 104773. [CrossRef]
11. Khan, M.U.; Sartaj, H.; Iqbal, M.Z.; Usman, M.; Arshad, N. AspectOCL: Using aspects to ease maintenance of evolving constraint specification. *Empir. Softw. Eng.* **2019**, *24*, 2674–2724. [CrossRef]
12. Barisal, S.K.; Dutta, A.; Godboley, S.; Sahoo, B.; Mohapatra, D.P. MC/DC guided Test Sequence Prioritization using Firefly Algorithm. *Evol. Intell.* **2021**, *14*, 105–118. [CrossRef]
13. Suhail, S.; Malik, S.U.R.; Jurdak, R.; Hussain, R.; Matulevičius, R.; Svetinovic, D. Towards situational aware cyber-physical systems: A security-enhancing use case of blockchain-based digital twins. *Comput. Ind.* **2022**, *141*, 103699. [CrossRef]
14. Ami, A.S.; Cooper, N.; Kafle, K.; Moran, K.; Poshyvanyk, D.; Nadkarni, A. Why Crypto-detectors Fail: A Systematic Evaluation of Cryptographic Misuse Detection Techniques. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–26 May 2022; pp. 614–631. [CrossRef]
15. Canakci, S.; Delshadtehrani, L.; Eris, F.; Taylor, M.B.; Egele, M.; Joshi, A. DirectFuzz: Automated Test Generation for RTL Designs using Directed Graybox Fuzzing. In Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC) 2021, San Francisco, CA, USA, 5–9 December 2021; pp. 529–534. [CrossRef]
16. Aleman, J.L.M.; Agenjo, A.; Carretero, S.; Kosmidis, L. On the MC/DC Code Coverage of Vulkan SC GPU Code. In Proceedings of the 41st Digital Avionics System Conference, Portsmouth, VA, USA, 18–22 September 2022. [CrossRef]
17. Tatale, S.; Prakash, V.C. Automatic Generation and Optimization of Combinatorial Test Cases from UML Activity Diagram Using Particle Swarm Optimization. *Ing. Syst. d'Inform.* **2022**, *27*, 49–59. [CrossRef]
18. Avdeenko, T.; Serdyukov, K. Automated test data generation based on a genetic algorithm with maximum code coverage and population diversity. *Appl. Sci.* **2021**, *11*, 4673. [CrossRef]

19. Lemieux, C.; Inala, J.P.; Lahiri, S.K.; Sen, S. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. 2023, pp. 1–13. Available online: https://github.com/microsoft/codamosa (accessed on 14 March 2023).
20. Fadhil, H.M.; Abdullah, M.N.; Younis, M.I. Innovations in t-way test creation based on a hybrid hill climbing-greedy algorithm. *IAES Int. J. Artif. Intell.* **2023**, *12*, 794–805. [CrossRef]
21. Gupta, N.; Sharma, A.; Pachariya, M.K. Multi-objective test suite optimization for detection and localization of software faults. *J. King Saud Univ.-Comput. Inf. Sci.* **2022**, *34*, 2897–2909. [CrossRef]
22. Khaleel, S.I.; Anan, R. A review paper: Optimal test cases for regression testing using artificial intelligent techniques. *Int. J. Electr. Comput. Eng.* **2023**, *13*, 1803–1816. [CrossRef]
23. Barisal, S.K.; Chauhan, S.P.S.; Dutta, A.; Godboley, S.; Sahoo, B.; Mohapatra, D.P. BOOMPizer: Minimization and prioritization of CONCOLIC based boosted MC/DC test cases. *J. King Saud Univ.-Comput. Inf. Sci.* **2022**, *34*, 9757–9776. [CrossRef]
24. Sartaj, H.; Iqbal, M.Z.; Jilani, A.A.A.; Khan, M.U. A Search-Based Approach to Generate MC/DC Test Data for OCL Constraints. In Proceedings of the Search-Based Software Engineering: 11th International Symposium, SSBSE 2019, Tallinn, Estonia, 31 August–1 September 2019; Lecture Notes in Computer Science. Springer International Publishing: Berlin, Germany, 2019; Volume 11664, pp. 105–120. [CrossRef]
25. Zafar, M.N.; Afzal, W.; Enoiu, E. Evaluating System-Level Test Generation for Industrial Software: A Comparison between Manual, Combinatorial and Model-Based Testing. In Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test, Pittsburgh, PA, USA, 17–18 May 2022; pp. 148–159. [CrossRef]
26. Jha, P.; Sahu, M.; Isobe, T. A UML Activity Flow Graph-Based Regression Testing Approach. *Appl. Sci.* **2023**, *13*, 5379. [CrossRef]
27. Tiwari, R.G.; Pratap Srivastava, A.; Bhardwaj, G.; Kumar, V. Exploiting UML Diagrams for Test Case Generation: A Review. In Proceedings of the 2021 2nd International Conference on Intelligent Engineering and Management (ICIEM), London, UK, 28–30 April 2021; pp. 457–460. [CrossRef]
28. Liu, Y.; Li, Y.; Deng, G.; Liu, Y.; Wan, R.; Wu, R.; Ji, D.; Xu, S.; Bao, M. *Morest: Model-Based RESTful API Testing with Execution Feedback*; Association for Computing Machinery: New York, NY, USA, 2022; Volume 2022-May. [CrossRef]
29. El-Attar, M.; Abdul-Ghani, H.A. Using security robustness analysis for early-stage validation of functional security requirements. *Requir. Eng.* **2016**, *21*, 1–27. [CrossRef]
30. Afrose, S.; Xiao, Y.; Rahaman, S.; Miller, B.P.; Yao, D. Evaluation of Static Vulnerability Detection Tools With Java Cryptographic API Benchmarks. *IEEE Trans. Softw. Eng.* **2023**, *49*, 485–497. [CrossRef]
31. Ribeiro, V.; Cruzes, D.S.; Travassos, G.H. Understanding Factors and Practices of Software Security and Performance Verification. In Proceedings of the 19th Brazilian Symposium on Software Quality, Sbcopenlib, Brazil, 1–4 December 2020.
32. Szűgyi, Z.; Porkoláb, Z. Comparison of DC and MC/DC Code Coverages. *Acta Electrotech. Inform.* **2013**, *13*, 57–63. [CrossRef]
33. Marques, F.; Morgado, A.; Fragoso Santos, J.; Janota, M. TestSelector: Automatic Test Suite Selection for Student Projects. In Proceedings of the Runtime Verification: 22nd International Conference, RV 2022, Tbilisi, GA, USA, 28–30 September 2022; Lecture Notes in Computer Science. Springer International Publishing: Berlin, Germany, 2022; Volume 13498, pp. 283–292. [CrossRef]
34. Senjyu, T.; Mahalle, P.N.; Perumal, T.; Joshi, A. *ICT with Intelligent Applications*; Springer: New York, NY, USA, 2020; Volume 1.
35. Yang, Y.; Xia, X.; Lo, D.; Grundy, J. A Survey on Deep Learning for Software Engineering. *ACM Comput. Surv.* **2022**, *54*, 1–73. [CrossRef]
36. Elyasaf, A.; Farchi, E.; Margalit, O.; Weiss, G.; Weiss, Y. Generalized Coverage Criteria for Combinatorial Sequence Testing. *IEEE Trans. Softw. Eng.* **2023**, 1–12. [CrossRef]