UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Software Failure Prediction Based on Patterns of Multiple-Event Failures

Caio Augusto Rodrigues dos Santos

Uberlândia, Minas Gerais
2021

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Software Failure Prediction Based on Patterns of Multiple-event Failures

Caio Augusto Rodrigues dos Santos

Tese de doutorado apresentada ao Programa de Pós-Graduação da Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Prof. Dr. Rivalino Matias Júnior

Uberlândia, Minas Gerais
2021

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

S237
2021

Santos, Caio Augusto Rodrigues dos, 1989-
Software Failure Prediction Based on Patterns of Multiple-Event Failures [recurso eletrônico] / Caio Augusto Rodrigues dos Santos. - 2021.

Orientador: Rivalino Matias Júnior.
Tese (Doutorado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.te.2021.202
Inclui bibliografia.
Inclui ilustrações.

1. Computação. I. Matias Júnior, Rivalino ,1971-, (Orient.). II. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. III. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:

Gizele Cristine Nunes do Couto - CRB6/2091

ACKNOWLEDGEMENT

I am deeply indebted to Prof. Rivalino Matias for providing me an opportunity to be a part of his research group. His priceless advice, support, and everlasting patience helped me finish this work.

I would like to thank the committee members - Prof. Kishor S. Trivedi, Prof. Artur Andrzejak, Prof. Dilma da Silva, Prof. Marcelo Keese Albertini - for their feedback and encouragement on this journey. I am especially grateful to Prof. Kishor S. Trivedi for providing me the opportunity to visit his research group.

Thanks to my friends and family for the fun times and for cheering me up during my gloomy days.

I also would like to thank the Federal University of Uberlândia and all its staff members.

Lastly, I am grateful to CAPES for the financial support.

RESUMO

Uma necessidade fundamental para a engenharia de confiabilidade de software é compreender como os sistemas de software falham, que significa entender a dinâmica que governa os diferentes tipos de manifestação de falha. Esta pesquisa apresenta um estudo exploratório sobre falhas de múltiplos eventos, que é uma manifestação de falha caracterizada por sequências de eventos de falha que variam em comprimento, duração e combinação de tipos de falha. Este estudo visa (i) melhorar a compreensão das falhas de múltiplos eventos em sistemas de software reais, investigando suas ocorrências, associações e causas; (ii) propor protocolos de análise que levem em consideração as manifestações de falha de múltiplos eventos; (iii) aproveitar a natureza sequencial desse tipo de falha de software para realizar previsões. As falhas analisadas nesta pesquisa foram observadas empiricamente. No total, foram analisadas 42.209 falhas reais de software de 644 computadores de diferentes locais de trabalho. As principais contribuições deste estudo são um protocolo desenvolvido para investigar a existência de padrões de associações de falha; um protocolo para descobrir padrões de sequências de falha; e uma abordagem de previsão cuja principal ideia é calcular a probabilidade de um determinado evento de falha ocorrer dentro de um intervalo de tempo após a ocorrência de um padrão particular de falhas anteriores. Três métodos foram utilizados para resolver o problema de previsão; Regressão Logística Multinomial (com ou sem regularização Ridge), Decision Tree e Random Forest. Tais métodos foram escolhidos devido à natureza dos dados de falha, nos quais os tipos de falha devem ser tratados como variáveis categóricas. Inicialmente, foi realizada uma análise de descoberta de associação de falhas que considerou apenas falhas de um sistema operacional (SO) comercial amplamente utilizado. Como resultado, foram descobertos 45 padrões de associação de falhas de sistema operacional com 153.511 ocorrências, compostos dos mesmos ou diferentes tipos de falha e ocorrendo, sistematicamente, em intervalos de tempo bem estabelecidos. As associações observadas sugerem a existência de mecanismos subjacentes que regem essas ocorrências de falha, o que motivou o aprimoramento do método anterior, com a criação de um protocolo para descobrir padrões de sequências de falhas usando limites de tempo flexíveis e uma abordagem de previsão de falha. Para ter uma visão abrangente de como as diferentes falhas de software podem afetar umas às outras, os dois métodos foram aplicados a três amostras diferentes — a primeira amostra contém apenas falhas do Sistema Operacional, a segunda contém apenas falhas de Aplicativos do Usuário e a terceira engloba falhas do Sistema Operacional e de Aplicativos de Usuário. Como resultado, foram encontradas 165, 480 e 640 sequências de falha diferentes com milhares de ocorrências, respectivamente. Por fim, a abordagem proposta foi capaz de prever falhas com boa até alta precisão (86% a 93%).

Palavras-chave: Falhas de software; associações de falha; sequências de falha; falhas de múltiplos eventos; padrões; predição.

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Software Failure Prediction Based on Patterns of Multiple-Event Failures

Caio Augusto Rodrigues dos Santos

Uberlândia, Minas Gerais
2021

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Ciência da Computação
Av. João Naves de Ávila, nº 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Tese de doutorado, 05/2021, PPGCO				
Data:	29 de março de 2021	Hora de início:	11:00	Hora de encerramento:	14:00
Matrícula do Discente:	11613CCP009				
Nome do Discente:	Caio Augusto Rodrigues dos Santos				
Título do Trabalho:	Software Failure Prediction Based on Patterns of Multiple-Event Failures				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Engenharia de Software				
Projeto de Pesquisa de vinculação:	-				

Reuniu-se, por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Marcelo Keese Albertini - FACOM/UFU, Kishor S. Trivedi - Duke University; Artur Andrzejak - Heidelberg University; Dilma Menezes da Silva - Texas A&M University e Rivalino Matias Júnior - FACOM/UFU orientador do candidato.

Os examinadores participaram desde as seguintes localidades: Kishor S. Trivedi - Durham, Carolina do Norte, Estados Unidos da América; Artur Andrzejak - Heidelberg, Baden-Württemberg, Alemanha; Dilma Menezes da Silva - College Station, Texas, Estados Unidos da América; Marcelo Keese Albertini e Rivalino Matias Júnior - Uberlândia/MG. O discente participou da cidade de Uberlândia/MG.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Rivalino Matias Júnior, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado.

Esta defesa faz parte dos requisitos necessários à obtenção do título de Doutor.

Ressalta-se que os examinadores Kishor S. Trivedi e Artur Andrzejak por serem estrangeiros, residentes em outro país e não possuírem CPF registrado no Brasil não podem assinar a ata de defesa.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Rivalino Matias Júnior**, **Professor(a) do Magistério Superior**, em 31/03/2021, às 16:00, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Marcelo Keese Albertini**, **Professor(a) do Magistério Superior**, em 31/03/2021, às 16:21, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Dilma Menezes Da Silva**, **Usuário Externo**, em 01/04/2021, às 15:32, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **2666991** e o código CRC **D2BD8F0A**.

Referência: Processo nº 23117.020718/2021-43 SEI nº 2666991

ABSTRACT

A fundamental need for software reliability engineering is to comprehend how software systems fail, which means understanding the dynamics that govern different types of failure manifestation. In this research, I present an exploratory study on multiple-event failures, which is a failure manifestation characterized by sequences of failure events, varying in terms of length, duration, and combination of failure types. This study aims to (i) improve the understanding of multiple-event failures in real software systems, investigating their occurrences, associations, and causes; (ii) propose analysis protocols that take into account multiple-event failure manifestations; (iii) take advantage of the sequential nature of this type of software failure to perform predictions. The failures analyzed in this research were observed empirically. In total, I analyzed 42,209 real software failures from 644 computers used in different workplaces. The major contributions of this study are a protocol developed to investigate the existence of patterns of failure associations; a protocol to discover patterns of failure sequences; and a prediction approach whose main concept is to calculate the probability of a certain failure event to occur within a time interval upon the occurrence of a particular pattern of preceding failures. I used three methods to tackle the prediction problem; Multinomial Logistic Regression (w/ and w/o Ridge regularization), Decision Tree, and Random Forest. These methods were chosen due to the nature of the failure data, in which the failure types must be handled as categorical variables. Initially, I performed a failure association discovery analysis which only included failures from a widely used commercial off-the-shelf Operating System (OS). As a result, I discovered 45 OS failure association patterns with 153,511 occurrences, which were composed of the same or different failure types and occurring within well-established time intervals, systematically. The observed associations suggest the existence of underlying mechanisms governing these failure occurrences, which motivated the improvement of the previous method by creating a protocol to discover patterns of failure sequences using flexible time thresholds and a failure prediction approach. To have a comprehensive view of how different software failures may affect each other, both methods were applied to three different samples — the first sample contained only OS failures, the second contained only User Application failures, and the third encompassed both OS and User Application failures altogether. As a result, I found 165, 480, and 640 different failure sequences with thousands of occurrences, respectively. Finally, the proposed approach was able to predict failures with good to high accuracy (86% to 93%).

Keywords: Software failures; failure associations; failure sequences; multiple-event failures; patterns; prediction.

CONTENTS

1 INTRODUCTION 1

1.1 CONTEXT 1

1.2 SIGNIFICANCE OF THE STUDY 3

1.3 RESEARCH QUESTIONS 5

1.4 HYPOTHESIS 5

1.5 GOALS 6

General: *6*

Specifics: *6*

1.6 OUTLINE 6

2 BACKGROUND 7

2.1 THEORETICAL FRAMEWORK 7

2.1.1 Software Failures *8*

2.1.2 Software Failures Analyzed *9*

2.1.3 Multinomial Logistic Regression *11*

2.1.4 Multinomial Logistic Regression with Ridge Regularization *13*

2.1.5 Decision Tree *14*

2.1.6 Random Forest *17*

2.1.7 Evaluating the Models Fit *18*

2.2 RELATED WORK 18

3 MATERIAL 22

3.1 INTRODUCTION 22

3.2 SOFTWARE UNDER STUDY 22

3.3 DATA COLLECTION 23

3.4 DATA CHARACTERIZATION 24

4 METHOD 30

4.1 INTRODUCTION 30

4.2 FAILURE ASSOCIATION DISCOVERY PROTOCOL 32

4.2.1 Taxonomy *33*

4.2.2 Protocol *34*

4.3 FAILURE SEQUENCE DISCOVERY PROTOCOL 39

4.4 FAILURE PREDICTION APPROACH 42

5 RESULTS 44

5.1 INTRODUCTION 44
5.2 OS FAILURE ASSOCIATIONS 45
5.2.1 Results Quality *48*
5.2.2 Effects of Changing the Search Intervals *50*
5.3 FAILURE SEQUENCES 52
5.3.1 Sequence Types *53*
5.3.2 Sequence Classes *56*
5.3.3 Sequence Lengths *57*
5.3.4 Sequence Durations *58*
5.4 FAILURE PREDICTION 61
5.4.1 Failure Prediction Based on Sequences of Two Failure Categories *61*
5.4.2 Failure Prediction Based on Sequences of Two Failure Types *66*
5.4.3 Failure Prediction Based on Sequences of Two Failure Types and the Δt between them *69*
5.4.4 Failure Prediction Based on Sequences of Three Failure Types *73*
6 CONCLUSION 76
6.1 INTRODUCTION 76
6.2 DEPENDENCE OF SOFTWARE FAILURE EVENTS 77
6.3 PATTERNS OF MULTIPLE-EVENT FAILURES 77
6.4 PREDICTION OF SOFTWARE FAILURES 79
6.5 THREATS TO VALIDITY 80
6.5.1 Internal Validity *80*
6.5.2 External Validity *81*
6.6 CONTRIBUTIONS TO THE LITERATURE 81
6.7 FUTURE WORK 82
REFERENCES 83
APPENDIX 91

LIST OF TABLES

TABLE 1.1. THE OCCURRENCE PROBABILITY OF MULTIPLE-EVENT FAILURE PATTERNS. 5
TABLE 3.1. COMPUTER USAGE PROFILE PER GROUP. 25
TABLE 3.2. FAILURES CATEGORIZATION. 25
TABLE 3.3. FAILURE CHARACTERIZATION PER GROUP. 26
TABLE 3.4. TYPES OF OS KERNEL FAILURES. 27
TABLE 3.5. CONSISTENT FAILURE TYPES. 29
TABLE 4.1. EXAMPLES OF FAILURE ASSOCIATION CANDIDATES. 33
TABLE 4.2. GROUPS WITH PFES AND/OR NFES FOR RF = FAILURE_A. 35
TABLE 4.3. THEORETICAL FAILURE ASSOCIATION CANDIDATES FOR RF = FAILURE_A AND LCE-FAILURE_A = {FAILURE_A, FAILURE_B}. 36
TABLE 5.1. OS FAILURE TYPES/SUBTYPES ACRONYMS USED IN THE RANKINGS. 45
TABLE 5.2. TOP FIVE ENTRIES IN RANKINGS OF THE CONFIGURATION FAC = PFE→RF. 45
TABLE 5.3. TOP FIVE ENTRIES IN RANKINGS OF THE CONFIGURATION FAC = RF→NFE. 46
TABLE 5.4. TOP FIVE ENTRIES IN RANKINGS OF THE CONFIGURATION FAC = PFE→RF→NFE. 46
TABLE 5.5. NUMBER OF FAILURE ASSOCIATION PATTERN OCCURRENCES PER CONFIGURATION. 50
TABLE 5.6. MOST CONSISTENT FAILURE SEQUENCES. 54
TABLE 5.7. MOST FREQUENT CAUSES IN THE FAILURE SEQUENCES. 55
TABLE 5.8. SEQUENCE CLASSES. 57
TABLE 5.9. TEN MOST FREQUENT SEQUENCE LENGTHS. 58
TABLE 5.10. SEQUENTIAL ASSOCIATIONS BASED ON FAILURE CATEGORIES. 62
TABLE 5.11. SUMMARY OF THE SEQUENCES' DURATIONS (IN HOURS) 69
TABLE 5.12. SUMMARY OF THE MODELS' ACCURACY. 70
TABLE 5.13. MODELS' ACCURACY (SEQUENCES OF THREE FAILURE TYPES). 73
TABLE A.1. REFERENCE FAILURES THAT APPEARED IN FOUR GROUPS. 91
TABLE A.2. REFERENCE FAILURES THAT APPEARED IN THREE GROUPS. 91

LIST OF FIGURES

FIGURE 1.1. AVOIDING FUTURE FAILURES BY EMPLOYING PREVENTIVE ACTIONS. 5

FIGURE 2.1. THE FUNDAMENTAL CHAIN OF DEPENDABILITY. 9

FIGURE 2.2. COMPARISON OF RELIABILITY CURVES. 10

FIGURE 2.3. EXAMPLE OF A *GINI* INDEX CALCULATION. 15

FIGURE 2.4. EXAMPLE OF A DECISION TREE. 16

FIGURE 2.5. EXAMPLE OF A RANDOM FOREST. 17

FIGURE 3.1. SYSTEMIC RELIABILITY. 23

FIGURE 3.2. RELIABILITY MONITOR. 24

FIGURE 3.3. AUTOMATIC FAILURE CATEGORIZATION. 26

FIGURE 3.4. DATASET STRATIFICATION 28

FIGURE 4.1. FAILURE ASSOCIATION VS FAILURE SEQUENCE. 31

FIGURE 4.2. SEARCHING FOR THE FAC = FAILURE_B→FAILURE_A. 36

FIGURE 4.3. PROTOCOL'S EXECUTION FLOW. 38

FIGURE 4.4. SEQUENCE DISCOVERY PROTOCOL. 40

FIGURE 4.5. FILTERING AND COLLECTING ΔTS BETWEEN FAILURES IN COMPUTER LOGS. 42

FIGURE 4.6. INPUT TO THE PREDICTION MODELS. 43

FIGURE 5.1. OVERVIEW OF THE METHODS AND SAMPLES. 44

FIGURE 5.2. NUMBER OF FACS THAT REMAIN CONSTANT BY DECREASING THE SI. 51

FIGURE 5.3. NUMBER OF FACS THAT IS REDUCED BY DECREASING THE SI. 51

FIGURE 5.4. NUMBER OF FACS THAT IS INCREASED BY DECREASING THE SI. 52

FIGURE 5.5. HISTOGRAM OF FAILURE SEQUENCE DURATIONS. 59

FIGURE 5.6. SEQUENCE LENGTH VS TIME DURATION 60

FIGURE 5.7. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE CATEGORIES (MULTINOMIAL LOGISTIC REGRESSION AND DECISION TREE). 64

FIGURE 5.8. DECISION TREE BASED ON SEQUENCES OF TWO FAILURE CATEGORIES. 64

FIGURE 5.9. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES (MULTINOMIAL LOGISTIC REGRESSION WITH RIDGE - SAMPLE_ALL). 67

FIGURE 5.10. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES (DECISION TREE - SAMPLE_ALL). 67

FIGURE 5.11. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES (RANDOM FOREST - SAMPLE_ALL). 68

FIGURE 5.12. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES AND THE ΔT BETWEEN THEM (MULTINOMIAL LOGISTIC REGRESSION WITH RIDGE - SAMPLE_OS). 70

FIGURE 5.13. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES AND THE ΔT BETWEEN THEM (DECISION TREE - SAMPLE_OS) 71

FIGURE 5.14. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES AND THE ΔT BETWEEN THEM (RANDOM FOREST - SAMPLE_OS). 71

FIGURE 5.15. DECISION TREE BASED ON SEQUENCES OF TWO FAILURE TYPES AND THE ΔT BETWEEN THEM (SAMPLE_OS). 72

FIGURE 5.16. PROBABILITIES BASED ON SEQUENCES OF THREE FAILURE TYPES (MULTINOMIAL LOGISTIC REGRESSION WITH RIDGE – FILTERED SAMPLE_ALL). 74

FIGURE 5.17. PROBABILITIES BASED ON SEQUENCES OF THREE FAILURE TYPES (DECISION TREE – FILTERED SAMPLE_ALL). 74

FIGURE 5.18. PROBABILITIES BASED ON SEQUENCES OF THREE FAILURE TYPES (RANDOM FOREST - FILTERED SAMPLE_ALL). 75

FIGURE 6.1. EXAMPLE OF SOFTWARE FAILURES DEPENDENCY. 77

FIGURE A.1. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE CATEGORIES (RANDOM FOREST). 92

FIGURE A.2. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES (MULTINOMIAL LOGISTIC REGRESSION WITH RIDGE - SAMPLE_OS). 92

FIGURE A.3. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES (DECISION TREE - SAMPLE_OS). 93

FIGURE A.4. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES (RANDOM FOREST - SAMPLE_OS). 93

FIGURE A.5. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES (MULTINOMIAL LOGISTIC REGRESSION WITH RIDGE - SAMPLE_$USER_{APP}$). 94

FIGURE A.6. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES (DECISION TREE - SAMPLE_$USER_{APP}$). 94

FIGURE A.7. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES (RANDOM FOREST - SAMPLE_$USER_{APP}$). 95

FIGURE A.8. DECISION TREE BASED ON SEQUENCES OF TWO FAILURE TYPES (SAMPLE_OS) 95

FIGURE A.9. DECISION TREE BASED ON SEQUENCES OF TWO FAILURE TYPES (SAMPLE_$USER_{APP}$). 96

FIGURE A.10. DECISION TREE BASED ON SEQUENCES OF TWO FAILURE TYPES (SAMPLE_ALL). 97

FIGURE A.11. DECISION TREE BASED ON SEQUENCES OF TWO FAILURE TYPES AND THE ΔT BETWEEN THEM (SAMPLE_$USER_{APP}$). 98

FIGURE A.12. DECISION TREE BASED ON SEQUENCES OF TWO FAILURE TYPES AND THE ΔT BETWEEN THEM (SAMPLE_ALL). 99

FIGURE A.13. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES AND THE ΔT BETWEEN THEM (MULTINOMIAL LOGISTIC REGRESSION WITH RIDGE - SAMPLE_$USER_{APP}$). 100

FIGURE A.14. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES AND THE ΔT BETWEEN THEM (DECISION TREE - SAMPLE_$USER_{APP}$). 100

FIGURE A.15. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES AND THE ΔT BETWEEN THEM (RANDOM FOREST - SAMPLE_$USER_{APP}$). 101

FIGURE A.16. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES AND THE ΔT BETWEEN THEM (MULTINOMIAL LOGISTIC REGRESSION WITH RIDGE - SAMPLE_ALL). 101

FIGURE A.17. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES AND THE ΔT BETWEEN THEM (DECISION TREE - SAMPLE_ALL). 102

FIGURE A.18. PROBABILITIES BASED ON SEQUENCES OF TWO FAILURE TYPES AND THE ΔT BETWEEN THEM (RANDOM FOREST - SAMPLE_ALL). 102

FIGURE A.19. PROBABILITIES BASED ON SEQUENCES OF THREE FAILURE TYPES (MULTINOMIAL LOGISTIC REGRESSION WITH RIDGE - SAMPLE_OS). 103

FIGURE A.20. PROBABILITIES BASED ON SEQUENCES OF THREE FAILURE TYPES (DECISION TREE - SAMPLE_OS). 104

FIGURE A.21. PROBABILITIES BASED ON SEQUENCES OF THREE FAILURE TYPES (RANDOM FOREST - SAMPLE_OS). 104

FIGURE A.22. PROBABILITIES BASED ON SEQUENCES OF THREE FAILURE TYPES (MULTINOMIAL LOGISTIC REGRESSION WITH RIDGE - SAMPLE_$USER_{APP}$). 105

FIGURE A.23. PROBABILITIES BASED ON SEQUENCES OF THREE FAILURE TYPES (DECISION TREE - SAMPLE_$USER_{APP}$). 106

FIGURE A.24. PROBABILITIES BASED ON SEQUENCES OF THREE FAILURE TYPES (RANDOM FOREST - SAMPLE_$USER_{APP}$). 107

FIGURE A.25. PROBABILITIES BASED ON SEQUENCES OF THREE FAILURE TYPES (MULTINOMIAL LOGISTIC REGRESSION WITH RIDGE - SAMPLE_ALL). 108

FIGURE A.26. PROBABILITIES BASED ON SEQUENCES OF THREE FAILURE TYPES (DECISION TREE - SAMPLE_ALL). 109

FIGURE A.27. PROBABILITIES BASED ON SEQUENCES OF THREE FAILURE TYPES (RANDOM FOREST - SAMPLE_ALL). 110

FIGURE A.28. DECISION TREE BASED ON SEQUENCES OF THREE FAILURE TYPES (SAMPLE_OS). 111

FIGURE A.29. DECISION TREE BASED ON SEQUENCES OF THREE FAILURE TYPES (SAMPLE_$USER_{APP}$). 112

FIGURE A.30. DECISION TREE BASED ON SEQUENCES OF THREE FAILURE TYPES (SAMPLE_ALL). 113

LIST OF ACRONYMS

ABB	ASEA Brown Boveri
BOINC	Berkeley Open Infrastructure For Network Computing
CART	Classification and Regression Trees
CI	Confidence Interval
CI_{MD}	Confidence Interval around the median
DLL	Dynamic-link library
ERP	Enterprise Resource Planning
EXP	Explorer.exe
FA	Failure Association
FAC	Failure Association Candidate
GUI	Graphical User Interface
HW	Hardware
IBM	International Business Machines Corporation
I/O	Input/Output
ID	Identification
IE	Internet Explorer
IEU	Internet Explorer Update
IT	Information Technology
IV	Independent Variable
LCE	List of Candidate Events
lFAC	Lists of Failure Association Candidates
LLNL	Lawrence Livermore National Laboratory
LPS	List of Potential Sequences
lrpn	List containing all occurrences of the RF researched and its PFEs and/or NFEs
macOS	Macintosh Operating System
MD	Median
MLR	Multinomial Logistic Regression
MS	Microsoft
N/A	Not Applicable
NetU	.Net Framework Update
NFE	Next Failure Event
OffU	Microsoft Office Update

OS	Operating System
OS_{APP}	OS Application
OS_{KNL}	OS Kernel
OS_{SVC}	OS Service
PFE	Previous Failure Event
RAC	Reliability Analysis Component
R_{APP}	Reliability of the User Applications layer
RF	Reference Failure
R_{HW}	Reliability of the Hardware layer
RM	Reliability Monitor
R_{OS}	Reliability of the OS layer
RPC	Remote Procedure Call
R_{System}	Reliability of the System
Sample_ALL	Sample that contains both OS and User Application failures
Sample_OS	Sample that contains OS failures.
$Sample\_USER_{APP}$	Sample that contains User Application failures
SD	Standard Deviation
SDF	SQL Compact Edition Database File
SI	Search Interval
SIDS	Sudden Infant Death Syndrome
SQL	Structure Query Language
SVM	Support Vector Machine
TG	Time Generated
UB	Upper Bound
USA	United States of America
$USER_{APP}$	USER Applications
UTC	Universal Time Coordinated
VSU	Visual Studio Update
WER	Windows Error Reporting
Win7	Microsoft Windows 7
WinU	Windows Update
WUS	*Windows Update Service*

NOTATION

$\boldsymbol{\beta}$	Vector of parameters of the Multinomial Logistic Regression (w/ and w/o Ridge regularization).
β_0	Intercept of the Multinomial Logistic Regression (w/ and w/o Ridge regularization).
$\boldsymbol{\beta}^T$	Transpose of the parameters vector of the Multinomial Logistic Regression with Ridge regularization.
Δt	Time difference between two failure events.
λ	Penalty applied to the parameters of the Multinomial Logistic Regression with Ridge regularization.
π	Probability of occurring a failure event.
F1→F2	The symbol "→" does not necessarily imply causation; this symbol indicates the temporal order of the failures. F1 is the first failure to occur followed by F2.
$F1^{hh:mm} \rightarrow F2^{hh:mm}$	The superscripts (hh:mm) indicate the respective failure times in hours (hh) and minutes (mm).
$F1^{Cause\_A} \rightarrow F2^{Cause\_B}$	The superscripts (Cause_A and Cause_B) indicate the respective failure causes. In Win7 they are usually shown as hexadecimal codes, but their format is platform-specific.
Gini index	Splitting criterion that is used to decide the order of the independent variables in a Decision Tree.
IV_*	Independent variable in the Decision Tree plot.
ln	Natural logarithm.
$\Pr(Y\|X)$	Probability of Y given X.
R^2	The McFadden's pseudo R^2 measures the amount of change by using the independent variables versus not using them.
s	Spearman's Rank correlation coefficient.
$X_{\Delta t}$	Represents the Δt between failures in sequences composed of two failure types, in the Multinomial Logistic Regression with Ridge regularization.
$\mathbf{x}_{\boldsymbol{\Delta t}}$	Vector of two dummy independent variables of the Δt in sequences composed of two failure types, in the Multinomial Logistic Regression with Ridge regularization. The first dummy variable refers to "Minimum ≤ Δt ≤ Median" and the second to "Median < Δt ≤ Maximum".
$\mathbf{x}_{\mathbf{b}}$	Vector of dummy independent variables of all possible failure types that immediately precede the last failure (dependent variable) of sequences composed of <u>two failure types</u>, in the Multinomial Logistic Regression with Ridge regularization.
$\mathbf{x}_{\mathbf{b1}}$	Vector of dummy independent variables of all possible failure

	types that immediately precede the last failure (dependent variable) of sequences composed of three failure types, in the Multinomial Logistic Regression with Ridge regularization.
$\mathbf{x_{b2}}$	Vector of dummy independent variables of all possible failure types that occurred previously to the X_{before_1} in sequences composed of three failure types, in the Multinomial Logistic Regression with Ridge regularization.
X_{Before}	Represents the failure event immediately preceding the last failure (dependent variable) of sequences composed of two failure types, in the Multinomial Logistic Regression with Ridge regularization.
X_{Before_1}	Represents the failure event immediately preceding the last failure (dependent variable) of sequences composed of three failure types, in the Multinomial Logistic Regression with Ridge regularization.
X_{Before_2}	Represents the failure event that occurred previously to the X_{before_1} in sequences composed of three failure types, in the Multinomial Logistic Regression with Ridge regularization.
x_{KNL}	Dummy independent variable of the OS_{KNL} category, indicating that occurred ($x_{KNL} = 1$) or not ($x_{KNL} = 0$) immediately before the last failure (dependent variable) of sequences composed of two failure categories, in the Multinomial Logistic Regression.
x_{svc}	Dummy independent variable of the OS_{SVC} category, indicating that occurred ($x_{SVC} = 1$) or not ($x_{SVC} = 0$) immediately before the last failure (dependent variable) of sequences composed of two failure categories, in the Multinomial Logistic Regression.
$x_{USERAPP}$	Dummy independent variable of the $USER_{APP}$ category, indicating that occurred ($x_{USERAPP} = 1$) or not ($x_{USERAPP} = 0$) immediately before the last failure (dependent variable) of sequences composed of two failure categories, in the Multinomial Logistic Regression.
Y	Dependent variable of the Multinomial Logistic Regression (w/ and w/o Ridge regularization).

1. INTRODUCTION

1.1 Context

Nowadays, it is indisputable that software has become the backbone of modern society. Critical systems, in areas such as energy and water supply, transportation, telecommunication, finance, and many others, are extensively dependent on software. This ubiquity of software makes the failures of such systems have a significant impact on human lives, causing from simple inconvenient to catastrophic damage (e.g., [1], [2]). Some examples of high impact software failures follow:

- Therac-25: Problems in the software responsible for controlling the Therac-25 radiation therapy machine caused a series of accidents between 1985 and 1987, in which patients were given overdoses of radiation, resulting in deaths or serious injuries [3].

- Ariane 5: In June 1996, the unmanned Ariane 5 rocket exploded forty seconds after its launch. The cost of the project was approximately $7 billion. The cause of the problem was a numerical conversion bug in the rocket's inertial control software [4].

- Knight Capital Group: In August 2012, the American company Knight Capital Group lost $440 million. Problems in the trading software from Knight Capital Group caused a flood of erroneous orders on the New York Stock Exchange, affecting both the company and the USA stock market [5].

- Boeing 737 Max: More than 300 people were killed after two crashes in October 2018 and March 2019 of the aircraft model 737 Max. These accidents were caused by failures on a stall-prevention software system that pushed the nose of the planes down. The 737 Max family of aircraft has been grounded by the USA since March 2019 and has cost Boeing and airlines billions of dollars [6], [7].

Note that the concern with the reliability of computer systems should not be exclusively on critical, specialized, and customized systems, since these systems very often use parts of general-purpose software, especially operating systems. For example, according to [8], critical USA Navy military navigation systems and ABB power plan IT systems run on top of regular Windows operating systems, which is a commercial off-the-shelf non-specialized software. In these examples, failures in the Windows operating system can impair the operation of critical applications that have high-reliability requirements. Another similar example is the wide use of

Linux operating systems on different non-critical and critical embedded systems (e.g., [9], [10]).

Nowadays, software failures, instead of hardware or human failures, are responsible for most of the computing systems downtime [11]. Because of the high dependence of society on software systems, ensuring the quality of the software has become a major requirement. We can measure the quality of software using attributes like functionality, usability, capability, maintainability, reliability, security, and performance. However, reliability is considered the principal quality factor among these attributes, since it takes into account software failures that are responsible for most of the unplanned inactivity of computer systems [12]. Formally, software reliability is defined as the probability of failure-free software operation for a given time, in a particular execution environment [13].

Given the increasing importance of the correct functioning of software systems, software reliability engineering emerges as a sub-discipline of system engineering that focuses on engineering techniques that try to quantify and guarantee the reliability of these systems. A fundamental need for software reliability engineering is to comprehend how software systems fail, which means understanding the dynamics that govern the manifestation of different types of failures that threaten software reliability. In general, software systems fail due to single-event failures or multiple-event failures [14]. The former is characterized by a lone failure event, which happens suddenly, with no other associated (predecessor/successor) failure events. On the other hand, the latter is characterized by a sequence of failure events, which varies in terms of length, duration, and combination of failure types. In both types of failure manifestation, their practical effects on the system vary from service degradation and malfunctioning to its hang or crash.

Reviewing the software reliability literature, we note that most of the empirical and experimental studies in this field have focused on single-event failures (e.g., [8], [15], [16], and [17]). In addition to investigating this type of failure manifestation, I believe that learning about multiple-event failure patterns is essential towards more effective and accurate software reliability approaches, being valuable from different theoretical and practical perspectives:

- Failure prediction/avoidance can be improved through the runtime detection of patterns of failure sequences.

- Software reliability assessment can benefit from analytical models that consider different types of failure associations – most of the current models assume that software failures are stochastically independent.

- The software testing process can be reduced by focusing efforts on failures that initiate sequences of failures.

Hence, in this research, I present an exploratory study on multiple-event failures. This study aims to (i) improve the understanding of multiple-event failures in real software systems, investigating their occurrences, associations, and causes; (ii) propose analysis protocols that take into account multiple-event failure manifestations; (iii) take advantage of the sequential nature of this type of

software failure to perform predictions. The failures analyzed in this research were observed empirically. To have a complete comprehensive view of how different software failures may affect each other, I investigated the failure behavior of a commodity operating system (OS) and User Applications running on top of it. It is important to point out that the proposed approach is not dependent on any specific failure data type, so it also can be applied to failure data from any other software systems, with minor changes.

1.2 Significance of the Study

Since the reliability of software systems has become a major requirement, several analytical models have been proposed to evaluate and quantify software reliability [18]. We can summarize these models as either black-box or white-box. The black-box models estimate the reliability of the software through its failure history, assuming a parametric model based on the time between failures or the number of failures over a specific time interval [17]. From another perspective, the white-box models estimate reliability through the representation of the software structure and are, for the most part, restricted to the operational phase of the software life cycle [19].

According to [20]-[22], a common characteristic among the majority of these reliability models is to assume that software failures occur independently. Although this assumption may interfere, significantly, with the results of the software reliability assessment, it is often used to adapt analytical models to mathematically treatable forms and simplify the estimation calculation of the model parameters. The problem is that this assumption of independence can lead to important consequences when the events of interest are not independent.

For example, in 2004, the British government announced that 258 murder trials would be reviewed given the improper use of this assumption [23]. These trials occurred in the context of sudden infant death syndrome (SIDS) or "cot death" in Britain, which refers to the death of healthy infants. These are mysterious circumstances that post-mortem exams cannot distinguish natural deaths from parental negligence or abuse. Therefore, based on Meadow's law, advocated by the prosecution witness Professor Sir Roy Meadow, that "one cot death is a tragedy, two cot deaths is suspicious and, until the contrary is proved, three cot deaths is murder", many innocent parents were sent to jail. This law assumes that cot deaths are entirely random, however, these mysterious deaths could have been linked by some unknown factors, something genetic for example, which would increase the chance of a family that had suffered one cot death to suffer another [24].

To exemplify this problem in the context of software reliability, consider a software system application composed of three modules arranged reliability-wise in series, i.e., the failure of any module causes the failure of the whole application. Suppose that the first module (*m1*) failed, consequently, the whole system went down. However, whenever a module fails, it is restarted to get the system back up and running. The reliability of each module is assumed to be 0.995, 0.987, and 0.973, respectively, for a 100-hour mission time. The system reliability (R_s) for that mission time after a first *m1* failure is calculated from the reliability of each module (R_{m1}, R_{m2}, R_{m3}) as follows:

$$R_s = (R_{m1} \times R_{m2} \times R_{m3})$$

$$R_s = 0.995 \times 0.987 \times 0.973$$

$$R_s = 0.9555 \text{ or } 95.55\%$$

Now, suppose each time the system is restarted due to *m1* failure, the *m2* reliability is reduced by 20%. In this case, there is dependency among some modules. Therefore, the system reliability after a first *m1* failure can be calculated as:

$$R_s = (R_{m1} \times (R_{m2} - 0.20) \times R_{m3})$$

$$R_s = 0.995 \times (0.978 - 0.20) \times 0.973$$

$$R_s = 0.7532 \; or \; 75.32\%$$

Note that when failures of the modules are assumed to be independent, the estimated system reliability is 95.55%, otherwise, it is 75.32%. This example shows that the accuracy of the reliability estimate of a system can be affected, in a non-neglectable way, when considering or not the assumption of independence. Hence, in contrast with the common practice of assuming failure independence in software reliability modeling, I initially investigate the existence of failure dependence in software systems. For this purpose, I create a data-driven approach that looks for patterns of failure associations, which means to search for evidence of failure dependence. This approach uses field failure records collected from different computers and work environments, and it helps one to test whether the failure independence assumption holds for a given real software system being reliability-wise modeled.

Once patterns of failure associations are discovered for a given system under study, the assumption of failure independence does not apply anymore, and patterns of failure sequences can be characterized. For this purpose, I create a method to search for patterns of failure sequences focusing on understanding the underlying mechanisms governing the multiple-event failure behaviors. When these behaviors can be understood then online failure prediction becomes feasible, especially for cases in which the time between failures, in a sequence, is long enough to anticipate actions. Since the impact of failures on software systems can be massive, early prediction of their occurrence plays a significant role in reliability engineering. Therefore, I conclude this study by proposing a statistical prediction approach that estimates the failure occurrence probabilities based on patterns of failure sequences.

Figure 1.1 shows a scenario of multiple-event failures (failures A, B, and C), which can be proactively handled by the proposed approach once it knows the ongoing pattern of chained events. In this example, given that failure A is known to precede (→) failure B that in turn is also known to precede failure C, next time failure A is perceived by a failure detector, it can perform actions to prevent the occurrence of other failures expected to occur right after failure A (e.g., failures B and C).

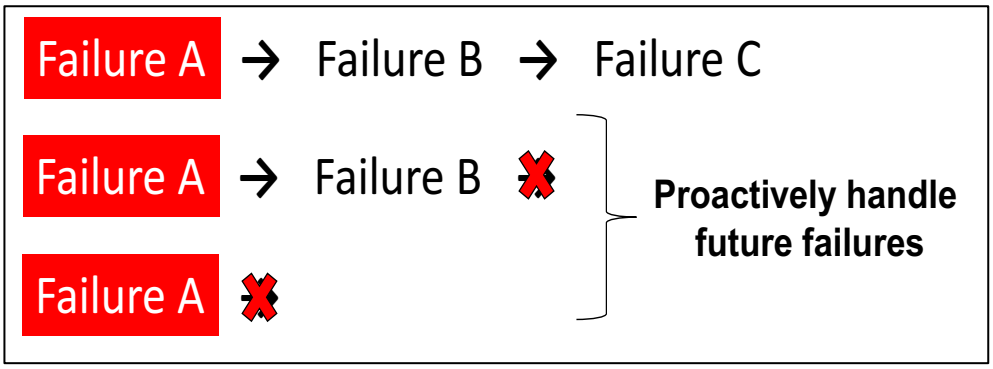


Figure 1.1. Avoiding future failures by employing preventive actions.

As can be observed, the symbol "→" indicates the temporal order of the failure occurrences belonging to the sequence analyzed and will be used throughout the document. Moreover, note that more than one sequence of failures can be initiated by the same failure type (Table 1.1), which requires analyzing the different occurrence probabilities of the possible sequences to make a decision. Once the next most probable event to occur is established, then proper preventive measures can be employed to prevent that failure type.

Table 1.1. The occurrence probability of multiple-event failure patterns.

Pattern	Probability
Failure A→Failure B→Failure C	70%
Failure A→Failure D	25%
Failure A→Failure E	5%

1.3 Research Questions

- Do software systems present systematic multiple-event failure manifestations?
    - If so, are these failure manifestations systematic enough to be considered patterns?
    - If so, could we use these patterns to predict future failures? At what level of accuracy? Would it be feasible to be implemented as part of an online failure prediction system?
- How OS and User Application failures affect each other?

1.4 Hypothesis

The first hypothesis tested in this study is that *real software systems may present systematic multiple-event failure manifestations*; which is a behavior that must be taken into account to fully assess the software reliability. Derived from this hypothesis, the second hypothesis tested in this study is that *patterns of multiple-event failures could be used to prevent future failures* given that we can take advantage of the sequential nature of these failures to perform prediction and preventive actions.

1.5 Goals

General:

- Investigate how to incorporate patterns of multiple-event failures in predictive models for software systems.

Specifics:

- Test the hypothesis of independence of failure events in real-world system failure data.
- Check how OS and User Application failures affect each other.
- Discover and examine patterns of software failure occurrences in real systems.

1.6 Outline

The remaining chapters of this work are organized as follows:

Chapter 2 presents the theoretical foundation that underpinned this study, covering theoretical concepts of software failures, the type of failure categorization adopted in this study, and the statistical methods used for failure prediction. Finally, it brings a review of the related literature.

Chapter 3 describes the sample of failure data used in this study. First, it explains the choice of the real-world commercial software product analyzed. Subsequently, I present the failure data collection method used, how the failure records were characterized, and finally a descriptive statistical analysis of the whole dataset.

Chapter 4 introduces the protocol created to discover patterns of failure associations, along with the taxonomy proposed to define various concepts used in the protocol. Besides, this chapter describes the algorithm developed to discover patterns of failure sequences. It concludes with the proposed prediction approach based on failure sequence patterns.

Chapter 5 presents the results of this research. Based on an empirical exploratory study, I evaluate the hypotheses presented in Chapter 1, as well as assess the methods described in Chapter 4. These methods are applied to the failure dataset detailed in Chapter 3, and I discuss the most relevant results and findings observed.

Finally, Chapter 6 presents final considerations on this study, highlighting the main results obtained, my conclusion, the threats to the validity of the research, the contributions to the literature, and the next steps planned for this research.

2. BACKGROUND

2.1 Theoretical Framework

The software failure taxonomy adopted in this study was defined in [25], which has broad acceptance by the software reliability community. In addition to the failure taxonomy, in [25] the authors present the main definitions related to the dependability of computer systems, which is a general concept that encompasses different system quality attributes. In this context, dependability is concerned with the confidence one can have in the correct system operation. Therefore, the dependability of a computer system is its ability to provide a service in which its users can justifiably have confidence in its operation. The dependability attributes defined in [25] are:

- **Availability**: readiness for correct service.
- **Reliability**: continuity of correct service.
- **Safety**: the absence of catastrophic consequences on the user(s) and the environment.
- **Integrity**: the absence of improper system alterations.
- **Maintainability**: ability to undergo modifications and repairs.

This study focuses on the **Reliability** of software systems. I use one of the main elements adopted to assess the reliability of a system, its failures. According to [26], the failure time is the most important variable when it comes to reliability. Both the black-box analytical models (e.g., [16], [18], and [27]) and the models using the white-box approach (e.g., [22], [28]) require information about the system failure behaviors. Therefore, it is essential to understand how failures occur in a system to allow the assessment of its reliability and thus to fully represent the real perception of its dependability.

Section 2.1.1 revisits the core concepts of software failures and how they occur in general, both notions introduced by the seminal work presented in [25].

Section 2.1.2 presents the particular semantics of operating system failures as considered in this study.

Sections 2.1.3, 2.1.4, 2.1.5, 2.1.6 describe the statistical models which I rely on to predict future failures based on patterns found in field data.

Section 2.1.7 presents how the statistical models are evaluated.

Finally, in Section 2.2, I present and discuss the related work.

2.1.1 Software Failures

To define terms such as software errors and failures, firstly it is necessary to present some basic concepts described in [25] to aid the understanding of these terms. These concepts are:

- **System:** is an entity that interacts with other entities, i.e., other systems, including hardware, software, humans.
- **Environment:** other systems that interact with a particular system are referred to as the environment of that system.
- **System boundary:** is the common frontier between the system and its environment.
- **System function:** is what the system is intended to do.
- **System behavior:** is what the system does to implement its function.
- **User**: is another system that receives service from the system provider.
- **Service:** the service delivered by a provider system is its behavior as it is perceived by its user.
- **Service interface:** part of the provider's system boundary where service delivery takes place.
- **Total state:** is the set of the following states: computation, communication, stored information, interconnection, and physical condition.
- **External and internal state:** the external state is the part of the provider's total state that is perceivable at the service interface; the remaining part is the internal state. The delivered service is a sequence of the provider's external states.
- **Use interface:** the interface of the user at which the user receives the service delivered by the provider system.

The **service** delivered by a **system** consists of a correct service when the service implements the **system function**, i.e., the service provided is in accordance with what the system is intended to do. A **failure** occurs when the delivered service does not comply with the functional specification, or when this specification does not adequately describe the function of the system. A system is considered to have failed when its output is received by the system user and is perceived as an unexpected or wrong result, i.e., the service provided does not follow the specified functional specification.

Since the delivered service is a sequence of **external states** of the system provider, a service failure means that at least one (or more) external system state(s) deviated from the correct service state. This deviation is called an **error**. Therefore, an error is the part of the system state that can lead to service failure.

Errors can be successively transformed into other errors; this transformation is called error propagation. Error propagation causes a system to fail if the error is propagated to the system service interface, causing the service to deviate from its specification, which is perceived by the system user as different from the expected result.

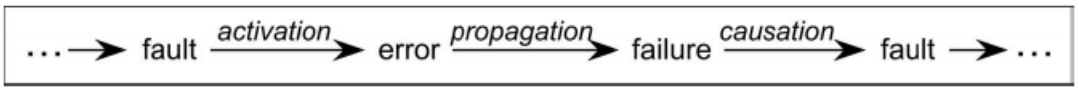


Figure 2.1. The fundamental chain of dependability.
Source: Avižienis et al. (2004) [25]

Although the perception of the error occurs at the time of service delivery, the system could have been in error long before. Therefore, failure is the perception by the system user of an error in the service provided. Many errors do not reach the external state of the system and therefore do not cause a failure.

The cause of an error is the activation of a **fault** (defect or bug). A fault is considered dormant until it is activated. Therefore, activating a fault causes an error, which may or may not reach the system service interface.

The three dependability threats (i.e., Fault, Error, and Failure) have a causal relationship called the fundamental chain of dependability. This chain is shown in Figure 2.1.

When a previously dormant fault is activated by a computation process, it is referred to as an **internal fault**. In this case, the execution of a given input causes the activation of the dormant fault. For example, suppose that a programmer incorrectly declares the type of a variable in the C programming language, declaring the variable as *int* instead of *long int*. In this case, the fault will lead to an error only when the variable reaches *long int* values, exceeding the *int* type storage capacity causing an overflow.

The activation of an **external fault** occurs when the failure of a system causes a fault (external) on another system (user) that receives the services of the failed provider system. This case is observed at the end of the fundamental chain of dependability representation (Figure 2.1).

The following is a description of the semantics of failures considered in this study.

2.1.2 Software Failures Analyzed

In this study, I analyzed two varieties of software failures — User Application and Operating System (OS) failures. The first refers to failures in any application that runs on top of the operating system that does not implement any OS functionality. The latter comprehends any failure in the operating system.

Specifically for the OS failures, I used an approach introduced in [29], which determines that if any component of the operating system does not deliver the correct service as specified for it, then there is an OS failure manifestation. Therefore, different from the common-sense notion of OS failure, in this study, an OS failure does not always result in a system hang/crash. This has to do with the fact that the malfunctioning of the failed OS component may not be severe enough to prevent the execution of the rest of the system; however, the OS parts that depend on the failed component may not run properly [30].

Note that studies on OS reliability (e.g., [16], [17], [27], [31], and [32]) have predominantly considered only OS failures in the Kernel space. However, as explained in [29], the assessment of an OS reliability by focusing on only Kernel-space failures is inaccurate since modern operating systems have several components running in both the Kernel and the User spaces. The components in the first space are critical OS subsystems that run in privileged mode and the ones in the second space are, in general, integrated OS programs that run in non-privileged mode. The OS components running in the User space perform their functions either in the foreground (e.g., OS programs like window manager) or in the background (e.g., OS services like print spooler). Widely used OS families like Windows, macOS, and Android, as well as new promising launches such as HarmonyOS [33], follow the similar architectural design in which the OS failure semantics above described and adopted in this study apply.

As mentioned above, an example of an OS component running in the User space and important for the system operation is the window manager application, which exists in GUI-based operating systems, such as the *explorer.exe* program [34] present in operating systems from the Windows OS family. Disrupting the operation of this program severely affects the user experience, for example, preventing users from accessing files and running programs. In this example, even if the OS Kernel is running correctly, from the user's point of view the OS failed [30].

Based on quantitative analyses, the studies in [29] and [35] demonstrate that OS reliability metrics computed considering failures in both levels (Kernel and User spaces) are more accurate than metrics that consider only failures in the Kernel space. Figure 2.2 compares the OS reliability considering only Kernel failures (traditional approach) with an approach that considers OS failures at both Kernel and User spaces. It becomes evident that following the traditional approach (dashed line) the OS reliability would be considered higher; nevertheless, the authors of [35] advocate that this result is not representative of the user experience since, from his or her perspective, the OS reliability is affected by any failure that causes the impairment or interruption of its features or services instead of only those confined to its Kernel space.

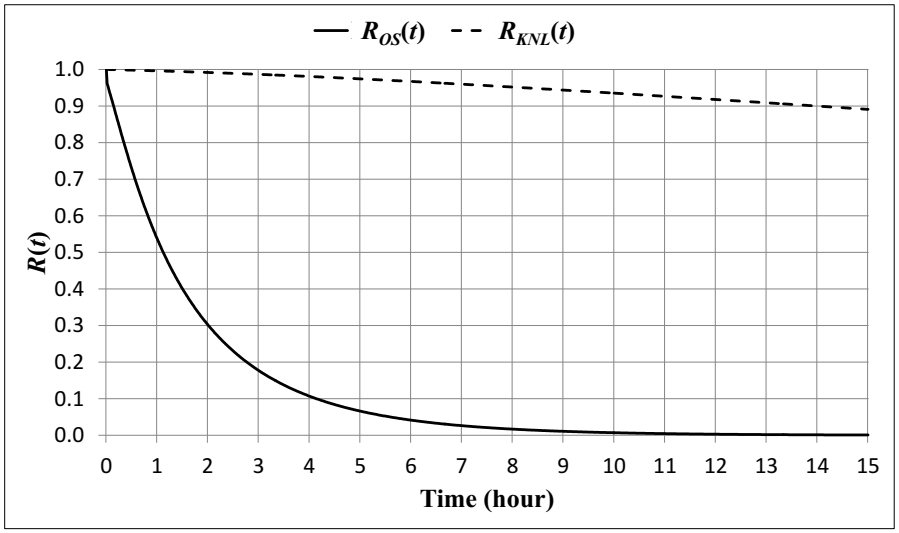


Figure 2.2. Comparison of reliability curves.
Source: Dos Santos et al. (2018) [35]

The authors in [35] state that the solid curve (see Figure 2.2) would better represent the OS reliability perception by the users, which can be observed in practice, for any OS, by comparing the reliability figures provided by the manufacturer (based on the traditional approach) with the actual users' quality of experience regarding the OS reliability. Therefore, following the approach introduced in [29], this study considered OS failures observed at both (Kernel and User) spaces. Moreover, the approach considers three categories of OS failures: OS Application (OS_{APP}), OS Service (OS_{SVC}), and OS Kernel (OS_{KNL}).

- **OS_{APP}:** encompasses failures caused by malfunctioning of OS applications that run in the User space and on user demand. The Win7 programs *msseces.exe* [36] and *explorer.exe* are examples of OS applications.
- **OS_{SVC}:** includes failures of OS service components, such as WER [37] and MsInstaller [38], that run in the background and with minimal or no user interaction.
- **OS_{KNL}:** contains failures of OS Kernel subsystems that typically cause the system to hang or crash.

A detailed description of how the failures were classified is presented in Chapter 3. The following sections describe the methods used to predict multiple-event failures.

2.1.3 Multinomial Logistic Regression

Like any other regression model, the goal of analyzing data with logistic regression is to find the best fitting and most parsimonious model that describes the relationship between a dependent variable and a set of independent variables [39]. Note that the logistic regression is used to model a binary or dichotomous dependent variable (e.g., Success = 0, Failure = 1) by computing the probability that the dependent variable takes one of the two possible values. The logistic regression function [39] is expressed as follows:

$$\pi = Pr(Y = 1 | x_1, \dots, x_p) = \frac{e^{\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p}}{1 + e^{\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p}}, \quad (1)$$

where π is the probability of occurrence of an event, β_0 is the intercept, β_1 to β_p are the regression parameters, and x_1 to x_p are the independent variables. Note that this function is nonlinear in its parameters. However, it can be linearized by the logit transformation [39]. Given π is the probability of an event occurring, the probability of the event not occurring is:

$$1 - \pi = Pr(Y = 0 | x_1, \dots x_p) = \frac{1}{1 + e^{\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p}}, \quad (2)$$

therefore, using the odds ratio (i.e., π / (1 - π)), one obtains:

$$\frac{\pi}{1 - \pi} = e^{\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p}. \quad (3)$$

Finally, taking the natural logarithm of both sides produces a linear function of the parameters:

$$ln\left(\frac{\pi}{1-\pi}\right) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p \,. \quad (4)$$

Equations 3 and 4 represent the logit transformation, in which the natural logarithm of the odds ratio is the logit. The importance of this transformation is that Equation 4 has many desirable properties of a linear regression model, like being linear in its parameters, the logit, *ln* (odds ratio), may be continuous and may range from -∞ to +∞, depending on the range of *x* [39].

I focused on a special case of Logistic Regression, the Multinomial Logistic Regression, which extends the simple Logistic Regression by allowing the dependent variable to assume more than two values; in this study, this multinomial regression is necessary due to the varying number of software failure types investigated. The Multinomial Logistic Regression produces a logit for every binary relationship. Since the dependent variable can assume any number of categorical values (with a minimum of three), one of its possible values is placed as a reference, and the other values are analyzed, in a binary fashion, with respect to the selected reference. The choice of the reference value may be arbitrary and changing it does not change the quality of the model; it only modifies the interpretation of the estimated parameters. Consequently, given k possible values of the dependent variable, $k - 1$ logits are created.

For example, if we wish to analyze the occurrences of one failure immediately after another (F1→F2), we will be analyzing the relationship of two categorical variables in which we want to predict F2 based on F1. Values *A*, *B*, and *C* are the three possible values (i.e., failure types) for the dependent variable (F2), and values *D*, *E*, and *F* for the independent variable (F1). Since the dependent variable assumes three levels, we need to create two logit functions, $g_A(\cdot)$ and $g_B(\cdot)$, in order to model this problem. In this example, *C* is used as the reference value,

$$g_A = ln\left(\frac{Pr(Y=A)}{Pr(Y=C)}\right), \quad g_B = ln\left(\frac{Pr(Y=B)}{Pr(Y=C)}\right).$$

Given that the independent variable is also categorical, it is necessary to use dummy variables to represent it. The dummy variables are binary variables that take values 0 or 1, which indicates if an observation does or does not contain the value that the variable represents. As they are binary, to represent n values, $n - 1$ dummy variables are needed. Thus, two dummy variables (e.g., x_D and x_E) are needed to represent the three levels of the independent variable in the example above. Hence, the resulting two logit functions are:

$$g_A = ln\left(\frac{Pr(Y = A|x_D,\ x_E)}{Pr(Y = C|x_D,\ x_E)}\right) = \beta_0 + \beta_1 x_D + \beta_2 x_E \quad (5)$$

$$g_B = ln\left(\frac{Pr(Y = B|x_D,\ x_E)}{Pr(Y = C|x_D,\ x_E)}\right) = \beta_0 + \beta_1 x_D + \beta_2 x_E \quad (6)$$

where x_D equals 1 if the independent variable assumes the value D, and 0 otherwise; and where x_E is 1 if the independent variable assumes the value E, and 0 otherwise. When both x_D and x_E are equal to zero, then it represents the case that the independent variable assumes the value F. Note that C is the reference value for the dependent variable in the logit functions. Thus, g_A is a comparison between A and C. In the same way, g_B is a comparison between B and C. The parameters (β_0, β_1, β_2) are estimated for both logits using the maximum likelihood technique. Subsequently, one can calculate the probability of each possible value of the dependent variable given the set of independent variable values [39]. The resulting conditional probabilities equations are:

$$Pr(Y = A|\, x_D, x_E) = \frac{e^{g_A}}{1 + e^{g_A} + e^{g_B}} \tag{7}$$

$$Pr(Y = b|\, x_D, x_E) = \frac{e^{g_B}}{1 + e^{g_A} + e^{g_B}} \tag{8}$$

$$Pr(Y = c|\, x_D, x_E) = \frac{1}{1 + e^{g_A} + e^{g_B}} \tag{9}$$

Equations 7, 8, and 9 provide the probabilities for all combinations of values between the dependent and independent variables, i.e., $Pr(Y = A$, or $Y = B$, or $Y = C \mid D$, or E, or $F)$. So, one of the main goals of this model is to predict the probabilities of occurrence of the values of the dependent variable given the values taken by the independent variables. In this study, the values of the dependent and independent variables are failure types selected from failure sequence patterns found in the system logs.

2.1.4 Multinomial Logistic Regression with Ridge Regularization

A challenge in using the Multinomial Logistic Regression is dealing with sparse data [40], i.e. when there is not enough data to cover all possible combinations between the dependent and independent variables. For example, suppose that in the example presented in Section 2.1.3, there is only log data of failures D occurring before failures A ($D \rightarrow A$), but not a single occurrence of D before B ($D \rightarrow B$).

Categorical variables, which are dealt with in this study, are often a challenge to sparsity, even in seemingly low-dimensional models (i.e., few independent variables), because typically at least one parameter is needed for each category (i.e., dummy variables) [41], [42].

Normally, the parameters ($\boldsymbol{\beta}$) of the Multinomial Logistic Regression are estimated using the Maximum Likelihood method. However, when the data is sparse, the method fails the estimation process [43]. To deal with this drawback of the Multinomial Logistic Regression, I had to incorporate a regularization method named Ridge [44]. The Ridge regularization adds a penalty to the estimation process, shrinking the parameters based on a penalty given by $\lambda * (\boldsymbol{\beta}^2)$ so that independent variables with a minor contribution to the outcome have their

parameters close to zero. The larger the value of λ, the greater the amount of shrinkage towards zero, but the parameters never reach zero. On the other hand, if λ equals zero, there is no penalty in the estimation process. I computed the best level of penalty using 10-fold cross-validation. To perform the regularization, I used the glmnet library [45], where the multinomial model of a dependent variable with k levels $Y = \{c_1, c_2, \ldots, c_k\}$ is given by:

$$Pr(Y = c_1 | X = \mathbf{x}) = \frac{e^{\beta_{0c_1} + \boldsymbol{\beta}_{c_1}^T \mathbf{x}}}{\sum_{l=c_1}^{c_k} e^{\beta_{0l} + \boldsymbol{\beta}_l^T \mathbf{x}}}$$

...

$$Pr(Y = c_k | X = \mathbf{x}) = \frac{e^{\beta_{0c_k} + \boldsymbol{\beta}_{c_k}^T \mathbf{x}}}{\sum_{l=c_1}^{c_k} e^{\beta_{0l} + \boldsymbol{\beta}_l^T \mathbf{x}}}$$

where, in this study, X represents patterns of preceding failures, $\mathbf{x}$ is a vector of dummy independent variables, β_0 is the intercept, $\boldsymbol{\beta}^T$ is the transpose of the regression parameters vector.

An equation is created for each level of the dependent variable. Note that the probability of each level is divided by the probability of all other levels. Essentially, for each level's equation (e.g., equations above), the Ridge regularization penalizes its parameters, in which the less relevant ones for the model are forced to be close to zero. In general, the Ridge helps on reducing the variance and thus making the model less sensitive to the training data [44].

2.1.5 Decision Tree

Similar to the Logistic Regression, Decision Trees are used to predict different levels (classes) of the dependent variable, considering the values taken by the independent variables [46]. As the name indicates, this method creates a tree structure that segments the training data by applying a series of decision rules. The tree is created in a top-down fashion (i.e., from the Root to the Leaf nodes). The Leaf nodes predict the outcomes of the dependent variable and each internal node acts as a test case for the independent variables where each edge corresponds to the possible answers to the test cases.

I adopted a well-known algorithm called CART [47] to build the decision trees in this study. The basic methodology is similar in all trees' algorithms, but their main differences rely on the tree structure, the splitting criteria, among other details. What stands out in the CART algorithm is the lower impact of outliers [48]. Moreover, the trees constructed by the CART algorithm only have binary splits, simplifying the splitting criterion that is used to decide the order of the independent variables in the tree (Internal nodes). This simplification of the splitting criterion makes it faster than the criterion used in the commonly adopted C4.5 algorithm [49]. The most common splitting criteria used in CART is named *Gini* index, which is defined as follows [49]:

$$Gini = 1 - \sum_{i=0}^{n-1} p_i^2 \tag{10}$$

where n indicates the number of levels (classes) of the dependent variable. c_i is the level (class) and s_i is the number of samples belonging to class c_i. Finally, $p_i = s_i/S$ is the relative frequency of class c_i in the data.

The first step to build a Decision Tree is to decide which independent variable will be the Root node. Therefore, for each of these variables, we calculate its *Gini* index. A lower value in the *Gini* index indicates a high segmentation of the training sample by the independent variable, placing it in higher positions of the tree. Figure 2.3 shows how the *Gini* index is calculated for the independent variable x_D in an example of failure sequences where one failure occurs immediately after another (F1→F2). The values *A* and *B* are the two possible values (i.e., failure types) for the dependent variable (F2), and values *D*, *E*, and *F* for the independent variable (F1). Similar to the Logistic Regression, it is necessary to use dummy variables to represent the categorical values of the independent variables.

Every independent variable acts as a decision in the tree. In the example depicted in Figure 2.3, the variable x_D indicates whether the previous failure is of type *D* or not (i.e., can be either *E* or *F*). Based on this decision, the training sample is divided into sequences that have and do not have the type *D* as the previous failure. Next, the *Gini* index is calculated for each subsample (D_{Yes} and D_{No}, respectively). If all occurrences of failure *D* happened before the same failure type, for example, type *A* (i.e., D→A), $Gini(D_{Yes})$ would reach its lowest value (zero). On the other hand, if 50% of sequences were composed of *D*→*A* and, consequently, the other 50% of *D*→*B*, $Gini(D_{Yes})$ would reach its highest value (0.5), indicating a poor segmentation of the training sample. Finally, after calculating the *Gini* index for both subsamples ($Gini(D_{Yes})$ and $Gini(D_{No})$), we can calculate the Total *Gini* index when using x_D to separate the training sample, which is calculated by the weighted average of both subsamples since they have a different number of values.

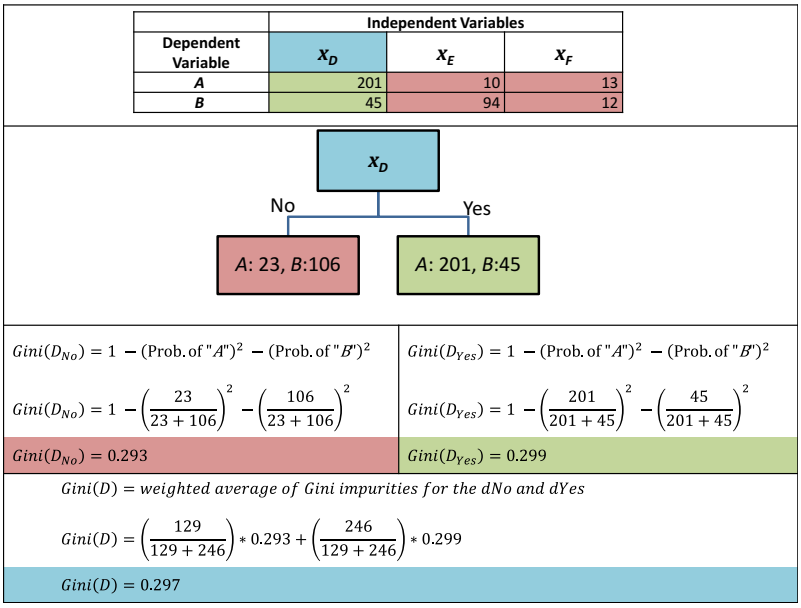


Figure 2.3. Example of a *Gini* index calculation.

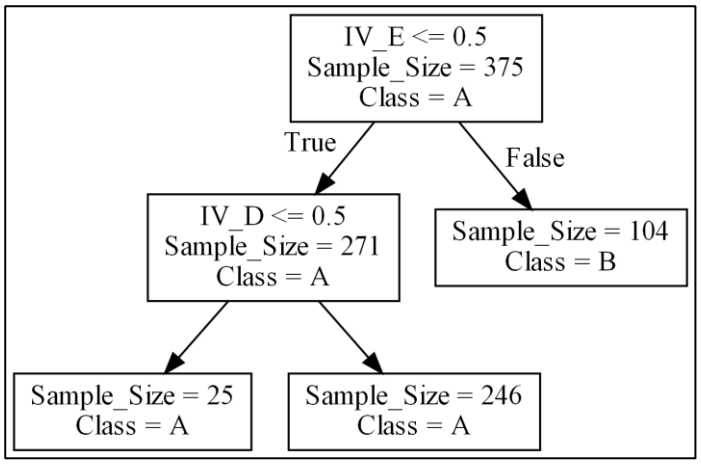


Figure 2.4. Example of a Decision Tree.

The Total *Gini* index of x_E ($Gini(E)$ = 0.288) and x_F ($Gini(F)$ = 0.317) are calculated in the same fashion. Next, the Total indexes are compared and the smallest (i.e., $Gini(E)$ = 0.288) is placed as the Root node. Figure 2.4 shows the resulting Decision tree. Note that both internal nodes (i.e., IV_E and IV_D) have three attributes:

1. **IV_* ≤ 0.5**: indicates the decision based on the independent variable (IV), which, in this case, is whether the previous failure is of a certain type in a sequence of two failures. For example, the decision in the Root node, i.e., IV_E ≤ 0.5, indicates that if the previous failure is not of type *E* (i.e., IV_E = 0) go to the left edge (IV_E ≤ 0.5 is True), and if the previous failure is of type *E* (i.e., IV_E = 1) go to the right edge (IV_E ≤ 0.5 is False). Note that the independent variables are either 0 or 1 since they are dummy variables.

2. **Sample_Size**: indicates the number of training data in the node. Therefore, the Root node contains the whole training sample. The other nodes segment the training data based on the decisions. For example, the internal node to the left of the Root node shows that 271 of the 375 sequences in the training data do not have the failure *E* as the previous failure.

3. **Class**: indicates the dependent variable value with the highest occurrence for the training data in a node. For example, the Root node shows that most sequences in the training sample have failure *A* as the last failure.

The only attribute Leaf nodes do not have is the first, which indicates the decision in the internal nodes. Moreover, the last attribute of the Leaf nodes (Class) indicates the predicted outcome of the dependent variable. Therefore, the Decision Tree shown in Figure 2.4 can be interpreted as a series of "ifs and elses":

- If the previous failure is *E*, the next failure is *B*
- Else if the previous failure is *D*, the next failure is *A*
- Else the next failure is *A*

Note that it is not necessary to include an internal node to represent failures of type *F*, because, in this example, if the previous failure is neither *E* nor *D*, it is of type *F*.

2.1.6 Random Forest

Decision Trees are considered as easy to build and interpret models (if the tree is small). Therefore, it emerged as one of the most adopted methods. However, the Decision Tree models suffer in terms of accuracy [42]. In other words, these models can have a great fit to the training sample, but they are ineffective when it comes to classifying new data.

Random Forest combines the simplicity of the Decision Tree with the flexibility to classify new data. It is part of ensemble methods, which, in this case, operates by constructing a collection of Decision Trees. These trees are generated from bootstrap samples from the training dataset. The classification occurs when the same input is applied to all trees and the most popular class is chosen [50]. Figure 2.5 presents an example of how a Random Forest is created and used. The red arrows indicate decision paths on every tree for a determined input.

It is important to point out that there is no rule of thumb to determine the number of trees on a Random Forest. According to [51], Radom Forests should have a number between 64 and 128 trees. However, this number may change due to the characteristics of the training data. Therefore, I evaluate 7 setups, starting by the default value 10 and ending in 128 (i.e., 10, 29, 49, 69, 88, 108, and 128), in every Random Forest model created and used the most accurate one.

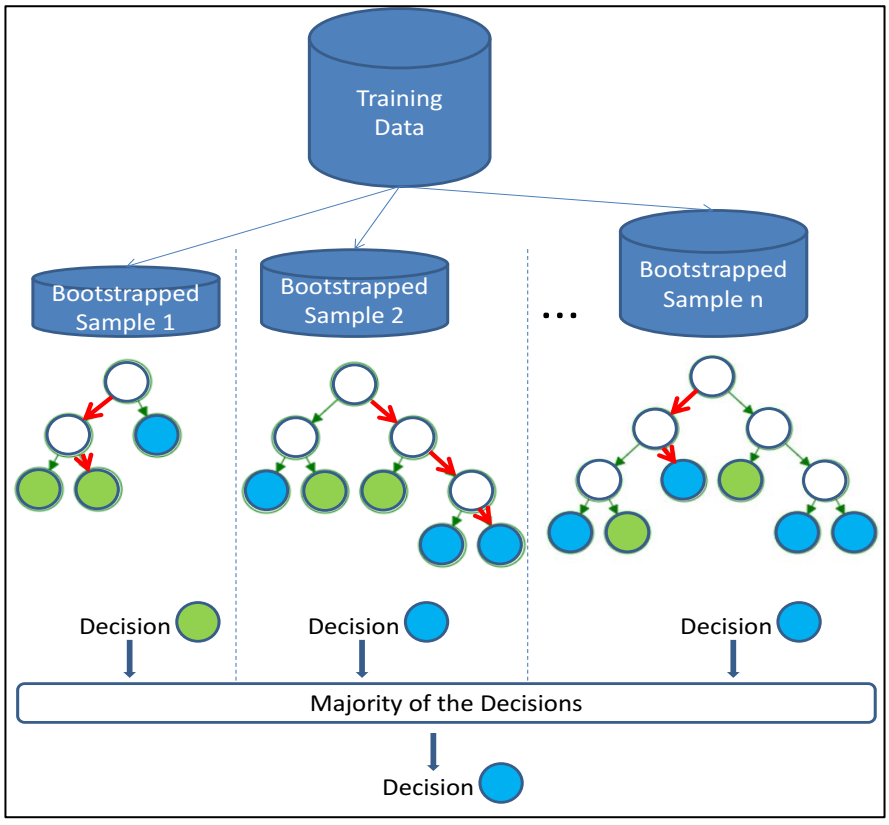


Figure 2.5. Example of a Random Forest.

2.1.7 Evaluating the Models Fit

Exclusive for the Multinomial Logistic Regression models without Ridge regularization I checked how well the models fitted the data using the coefficient of determination (R^2), which indicates the variance in the dependent variable that is predictable from the independent variable(s) [52]. There are many different ways to calculate the R^2 for Logistic Regression. Unfortunately, there is no consensus on which R^2 measure is the best. According to [53] and [54], the most prominent one and most often reported in statistical software is the McFadden's pseudo R^2 [55]. Therefore, I choose it to compute the models' goodness of fit. The McFadden's pseudo R^2 measures the amount of change by using the independent variables versus not using them. Thus, it indicates the difference of fit between using the model relative to using no model at all [54]. The pseudo $R^2 < 0.05$ indicates poor fit and the pseudo $R^2 > 0.20$ indicates a very good fit. A pseudo $R^2 > 0.40$ is rarely observed [53]. The maximum value (pseudo $R^2 = 1$) is a theoretical value that implies that all predicted probabilities would either be one value or the other in the binary relation of the logit; which is impossible to fit with a logistic curve, given that multinomial logistic models assume that the dependent variable cannot be perfectly predicted in any case [53].

For all models, I adopted a cross-validation approach to measuring how accurate the model is to predict the dependent variable based on the independent variables. In order to calculate this measurement, I split the work sample into training and test samples. To decrease the bias of selecting between these two data sets, I randomly split the data as follows: training (70%) and test (30%). I estimate the models based on the training set and test their accuracy against the test set, which contains data unseen during the model fitting phase; this approach provides an unbiased evaluation of the models' accuracy. The model's accuracy is defined as the number of correctly predicted values of the dependent variable with respect to the total number of sequences in the test set.

2.2 Related Work

Looking at the software reliability literature, especially related to operating system reliability, it can be seen that most of the studies are based on failure logs. Hence, next, I review works that proposed models for estimating software reliability by taking into account the dependency of failure events. Subsequently, I analyze works that proposed approaches to handle sequential pattern analysis. Finally, I discuss studies that proposed failure prediction approaches.

The authors in [27] collected and analyzed 1,546 Microsoft Windows XP failures from 200 computers from an academic environment. To collect the failure data, they used Microsoft's Corporate Error Reporting software. The authors concluded that OS failures were much less frequent than User Application failures. They conjecture that this prevalence can be explained by the interaction of applications, as well as the extensive use of dynamically linked library files that are invoked by multiple applications and are not robust enough.

In [16], the authors used the BOINC Crash Collector to collect and classify 2,528 Windows XP Kernel failures. BOINC is a platform for grouping resources

from volunteer computers to collect data and perform distributed calculations. As a result, the authors collected OS Kernel failures from 617 volunteer computers. Through failure data-mining, they found that poorly written device drivers caused more than 75% of the operating system failures.

Corroborating the results presented in [16], the authors in [17] also reported that device drivers were the main cause of OS Kernel failures in the systems they investigated. The authors pointed out that device drivers accounted for more than 70% of the Linux Kernel code at that time, and Windows XP introduced more than 35,000 different device drivers. They concluded that third-party extensions, at the Kernel space, were a major cause of failures of Windows XP, with device drivers accounting for 85% of reported failures.

Note that the three above-mentioned studies only considered Kernel-space failures as OS failures, which does not offer an accurate analysis of the OS reliability as a whole, since modern operating systems have parts running at both Kernel and User spaces [29]. In order to cover this issue, the authors in [56] used the OS failure classification proposed in [29], which considers not only Kernel failures as OS failures, but also OS application and OS service failures; these two other failure categories are observed at the User space. They found that OS service failures prevailed over OS Kernel and OS application failures, which had similar occurrence frequencies. Based on the statistical analysis of the OS failure times, the authors found that the density functions that best fitted the failure times were Gamma and Weibull.

All the above-cited works did not consider the dependence between OS failures, which can result in less accurate, or even erroneous, reliability assessments. As mentioned in Section 1.2, most software reliability models in the literature assume that software failures occur as statistically independent events, mainly to mathematically simplify the reliability analysis. As a result, the estimated reliability metrics obtained employing these models may differ significantly from the real reliability in cases where failures are not independent.

In [22], a software reliability-modeling framework that considers the dependency between failures was proposed. The framework uses the Markov renewal modeling approach [57], which is flexible enough to model failure dependency; but the large state space problem can be a serious drawback. According to the authors, to apply their framework in real scenarios, it is necessary to develop more detailed and specific models within the framework, as well as the use of statistical inference for the models' parameters. Such inferences are only possible based on field failure data. Other research works (e.g., [20], [28]) also proposed theoretical models to estimate software reliability incorporating failure dependence. However, none of them used empirical evidence to support or validate their models.

In [58], the authors showed that assessing the reliability of software in an artificial environment can produce an incomplete and imprecise estimate of its reliability due to the significant influence of the system environment. They highlight the importance of using real field data for reliability modeling purposes. Performing an empirical study on more than 200,000 computers running the Microsoft Windows operating system, they analyzed User Application failures and

found that the reliability of individual applications is affected by other applications installed on the system. Moreover, environmental factors such as the hardware and workload also influence the reliability of the applications analyzed. Therefore, the analysis in a single environment can result in a less accurate assessment. To mitigate the effect of a particular environment, in this study I consider as patterns only those failure combinations that regularly occurred in different computers from different workplaces.

In this study, I analyzed failure logs, which are, essentially, sequences of events ordered by their timestamp. Since sequence data is frequently observed in many fields, I searched for approaches that dealt with discovering sequential patterns. In the sequential pattern mining field, I found several approaches whose task is to find all frequent subsequences in a sequence database. In this case, a subsequence is classified as a frequent or a sequential pattern if it occurred in no less than a user-specified threshold [59]. The most popular approaches are GSP [60], Spade [61], PrefixSpan [62], Spam [63], and Lapin [64]. All these methods take as input a sequence dataset and the user-specified frequency threshold. These approaches only differ on how they discover the sequential patterns. Therefore, they should have the same output when running with the same input [59]. Besides the need for a threshold specified by the user, these methods allow gaps between consecutive events [65]. For example, if a sequence dataset has three sequences: S_1 = {CAGAAGT}, S_2 = {ACAGT}, and S_3 = {GAAGT} and threshold of three (i.e., sequential patterns should be present in each sequence), then the subsequence {AT} is considered a pattern although these two events are not consecutively after each other in the sequences. As a consequence of allowing gaps between consecutive events, these methods are not suited for this work because gaps may lead to erroneous decisions when analyzing distant (in time) failure events as if they happened together, which is undesired in the problem under study.

In the search for a new approach to discover sequence patterns without gaps between consecutive events, I found the substring mining field. The most common approach to mine substrings is called suffix tree [65], which, as the name suggests, creates a tree of suffixes for a string. Other strings can be incorporated into the suffix tree of a string and the frequency of a certain suffix (substring) occurring in multiple strings can be calculated. In this study, the suffixes would be considered as subsequences of failure within the failure logs (dataset), which would be treated as sequences. However, the timestamp of the events is neglect in both substring and sequential pattern mining. Differently, in my proposed approach, the timestamp is essential to the analysis because it provides information about the consistency of the associations. Moreover, I ensured that the associations found were patterns that repeated themselves in various groups of the dataset since I am interested in only systematic associations. It is important to point out that, after searching the literature, I did not find well-known methods that could be used to discover patterns of failure sequences, which ruled out the possibility of comparison analyzes between my approach and an alternative one.

Concerning failure prediction, the authors in [66] analyzed failure events of the IBM BlueGene/L supercomputer located at the Lawrence Livermore National Laboratory (LLNL). They analyzed more than a million events occurring in various system components, in which the failure events were extracted and categorized according to the subsystem in which they occur: memory, network, application

I/O, midplane switch, and node card failures. However, they did not describe the types of failures occurring in each subsystem, which is important to fully understand and characterize the behavior of such failures. The authors partitioned the time into fixed intervals and then tried to forecast whether there would be failure events in each interval based on the event characteristics of the preceding intervals. They could predict 80% of the memory and network failures, and 47% of the application I/O failures. Moreover, the authors observed that fatal failure events (i.e., failure events that caused system crash/hang) usually occurred after non-fatal events. However, they noticed that the time interval had a substantial impact on prediction accuracy. As this time decreased, the prediction difficulty increased rapidly, leading to degraded performance. Given this drawback, in this study, I adopted a data-driven approach in which a failure sequence is only considered as input to the prediction methods when the time between the failures composing it is similar and consistently observed on other occurrences of the same sequence in the dataset.

The authors in [67] used random indexing to represent the sequence of operations extracted from system logs and then applied Support Vector Machines (SVM) to separate normal and abnormal sequences. They reported that their predictor was able to almost perfectly detect non-failure conditions but was poor at identifying failure ones. They had better results, with most true positive rates higher than 0.50, by using a weighted SVM to account for this discrepancy by assigning a larger penalty for false negatives than false positives. However, I conjecture that it is infeasible to perform this analysis in large software systems because the system would have to store an enormous amount of both non-failure and failure events. Moreover, using non-failure events to predict failures could generate numerous false positives.

As can be observed in this section, in general, most empirical studies on software reliability do not consider the possible relationships between failure events. This work not only provides empirical evidence of dependency between failures but also investigates the nature of this dependency, that is, how different failures relate to each other. Different from the studies that proposed theoretical models to estimate software reliability based on the correlation between failures, but without using the information on actual failure data, the present study investigates these failure associations based on failures observed in real work environments. Moreover, this work presents a prediction approach using sequence patterns of multiple-event failures.

3. MATERIAL

3.1 Introduction

The purpose of this chapter is to describe the failure dataset analyzed in this study, as well as the details involved in its collection and how its failures are categorized. It is important to point out that the dataset's failure collection process was not part of this research, but it was performed during [29] and [56].

In Section 3.2, the software system under study is presented, as well as the justifications of its adoption in this study.

Section 3.3 describes the mechanism for collecting the failure data.

Finally, in Section 3.4, I explain the dataset-stratification approach adopted and present a detailed description of the work sample.

3.2 Software Under Study

In most computer systems, the software layer is divided into User Application software (or simply application) and Operating System software. Given the hierarchical nature of these two tiers, applications fully depend on the operating system (OS) to be executed [35]. Due to this technical dependence, OS failures can severely impair even the most reliable application software [30]. Therefore, if the OS layer does not guarantee, at least, the same level of reliability expected for the user-level application, the whole system reliability is compromised [56].

Figure 3.1 illustrates the above-mentioned scenario, in which the computer system's reliability as a whole decreases due to the lower OS reliability, although the other system's layers (hardware and applications) are highly reliable (99.999%). Despite the evident importance of an OS failure-free operation to help ensure system reliability, an extensive review of the software reliability literature shows a lack of research works targeting operating systems.

Given the importance and lack of research in this field, this empirical study initially focused on operating system software reliability. In particular, I assessed a commercial off-the-shelf operating system, given that nowadays this class of OS has been used in systems requiring high reliability [8]-[10]. Moreover, to have a holistic viewpoint on the reliability of the software layer, I also considered User Application failures ($USER_{APP}$). User Applications are usually installed by the user or are installed along with the OS installation but do not implement any OS

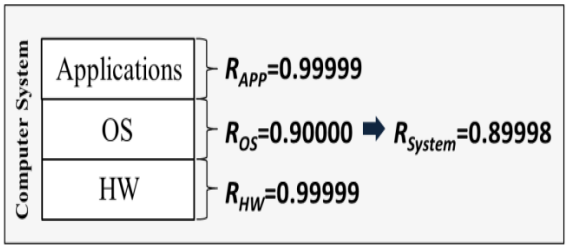


Figure 3.1. Systemic reliability.
Source: Dos Santos et al. (2018) [35]

functionality. The Microsoft Internet Explorer and Microsoft Notepad programs are examples of User Applications that are installed together with the Microsoft Windows 7 OS.

The field data used in this study were collected between the years 2010 and 2014 [56]. The analyzed dataset is composed of failure records collected from logs of computers running Microsoft Windows 7 (Win7) in different production environments. Note that, at the above-mentioned collecting period, Win7 was the most used commodity OS, and nowadays this is still one of the most used OS having more than 23% of the market share [68]. Moreover, I conjecture that most of the empirical findings revealed in this study may be analogous in other operating systems from the same OS family (e.g., Windows 8/8.1/10).

Due to its wide utilization in different work environments, Win7 was chosen as the target software of this study. The next section explains how Win7 logs and stores its failures and the mechanism that was used to collect them.

3.3 Data Collection

Windows 7 provides detailed failure records through its *Reliability Analysis Component* (RAC) [69]. The RAC stores Win7's failure data into a single file (SQL Compact Edition Database File) named *RacWmiDatabase.sdf*. The RAC collects all failure events on the computer since the OS installation date, and it is automatically enabled during the installation of the operating system. Then, it starts creating and storing a log record for every failure event in the computer [70], which contains ten fields as described below:

- ***ComputerName***: specifies the canonical name of the computer in which the failure record was generated.
- ***EventIdentifier***: identifies the failure event — a numerical value assigned to the source (application, service, or Kernel subsystem) that stored the failure record into the log file.
- ***InsertionStrings***: contains textual information (detail strings) that supplements the failure report.
- ***Logfile***: specifies the name of the event log file.
- ***Message***: specifies the event message as it appears in the event log file.
- ***ProductName***: specifies the product (program) name that is associated with the failure event.
- ***RecordNumber***: identifies the failure record entry within the event log file.

- ***SourceName***: specifies the name of the source (application, service, or Kernel subsystem) that generated the failure event.
- ***TimeGenerated***: specifies the time in UTC at which the source generated the event, that is, the moment that the failure event was registered by RAC.
- ***User***: specifies the username of the logged-on user when the failure event occurred.

The data collected by RAC is used in the Reliability Monitor (RM). RM is a tool that tracks hardware and software problems and other computer changes, providing the computer's stability index, which is a value from 0 to 10. The higher this index, the more stable the system is [71]. Figure 3.2 shows the main screen of the RM. The data collected by the RAC is visualized through a chart. Note that the circles represent the events and the line represents the stability index. It can be seen that the failure of the *explorer.exe* program, which occurred on 05/21/2012, directly affected the stability index. By clicking on a specific event one can get details about it.

Two methods were adopted for data collection [56]. Initially, the RAC files (.sdf) were manually copied from the surveyed computers, which required local access to the computers. For every file copied, a paper form was filled out with information regarding the system's usage profile and running applications, in order to characterize the surveyed computer. Complementarily to the onsite data collection, an online web page was also created to upload the RAC files remotely, where copies of the characterization form are completed online and assigned to the uploaded RAC files [72].

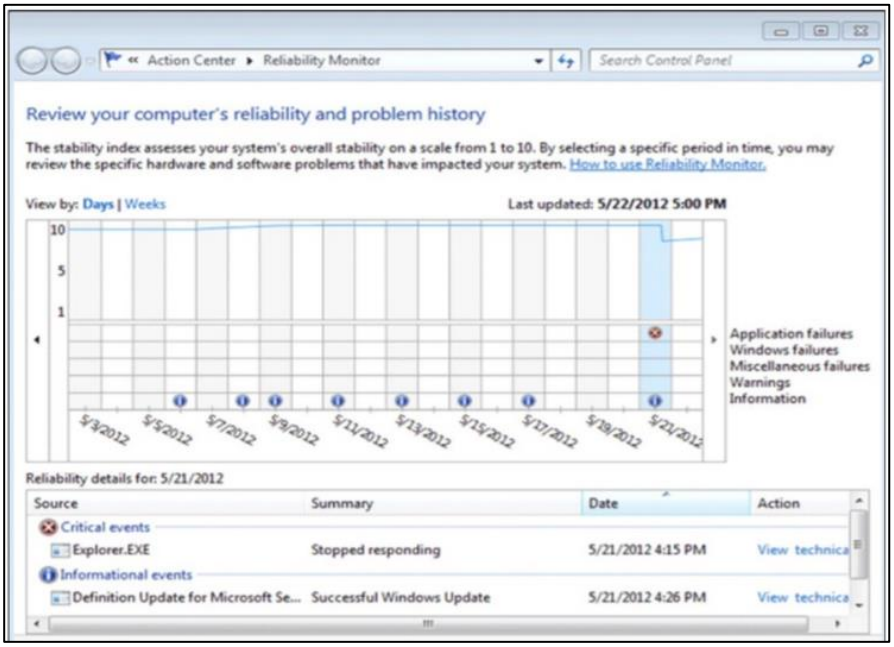


Figure 3.2. Reliability Monitor.

3.4 Data Characterization

In total, 644 computers were surveyed, which resulted in 42,209 software failures. The collected failure data were organized into four groups (G1 to G4). Failures in the first three groups were collected onsite, and failures in G4 were

collected through the online web form mentioned in Section 3.3. Table 3.1 shows the usage profiles of the four groups.

Groups G1 and G2 contain failures from computers used in a university campus. G1 contains failures from computers used in undergraduate teaching laboratories. These computers are mainly used by students to run applications like text and graphics editing, web and software development, and Internet browsing. G2 contains failures from computers used in the university's administrative offices; not related to teaching laboratories. Furthermore, the failures in G3 came from a corporate workplace.

The first three groups are homogeneous, i.e., each one contains failure data collected from computers located at the same workplace. On the other hand, G4 is a heterogeneous group, i.e., it contains failure data collected from computers located at different workplaces, varying from corporate, academic, and home/personal computers.

Initially, it was performed a manual classification of the failure events found in a pilot sample of the collected RAC files [56]. The manual classification relied on the fields *EventIdentifier*, *ProductName*, and *SourceName*. Table 3.2 presents an example of the failure categorization approach. Most failure events were categorized based on the fields *EventIdentifier* and *SourceName*. However, note that the first two failure events in Table 3.2 have the same *EventIdentifier* (1000) and *SourceName* (Application Error). Therefore, in this case, it also needed to analyze the field *ProductName*. Based on this further analysis, we know that the second event is related to the *chrome.exe* program, which is a User Application and not an OS Application. On the other hand, the first entry is related to *explorer.exe*, which is an OS (Win7) Application.

Table 3.1. Computer usage profile per group.

Group	Workplace	Application Profile
G1	Academic	Office apps, graphic editing, software & web development.
G2	Corporate	Office apps, graphic editing, software & web development.
G3	Corporate	Office apps, software & web development.
G4	Heterogeneous	Office apps, graphic editing, software & web development, multimedia, scientific & engineering, games, ERP apps., point of sale, antivirus, servers: database, web, application, e-mail, directory, and file/printer.

Table 3.2. Failures categorization.

Event Identifier	SourceName	ProductName	OS Failure Category
1000	Application Error	explorer.exe	OS Application
1000	Application Error	chrome.exe	N/A
1002	Application Hang	rundll32.exe	OS Application
1137	WindowsStartupRepair	Windows	OS Kernel
20	WindowsUpdate client	Windows Update	OS Service

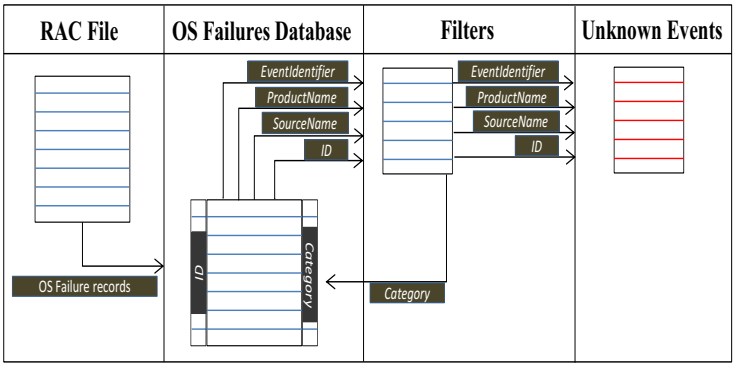


Figure 3.3. Automatic failure categorization.
Source: Dos Santos, Matias, and Trivedi (2019) [14]

Based on the pilot sample, filters were used to automate the process of classifying the failures of the whole dataset, based on script programming. Since the filters were initially created based on a pilot sample, they were refined progressively, incorporating new failure classification rules as the classification process evolved to cover the whole dataset.

Figure 3.3 gives an overview of the automatic categorization of the dataset. In addition to the fields *EventIdentifier*, *ProductName*, and *SourceName*, the script creates an ID field to identify each failure uniquely in the failure database, and a *Category* field that is filled out after the failure categorization. Next, the script checks whether the failure event, characterized by the fields *EventIdentifier*, *ProductName*, and *SourceName*, has the corresponding classification implemented by the filters. If the failure event is known, one of the four categories considered in this study (OS_{KNL}, OS_{SVC}, OS_{APP}, and $USER_{APP}$) is assigned to the *Category* field created for each failure event; otherwise, the script identifies and stores this failure event record into a list of unknown failure events, which have to be classified manually through a detailed analysis of all fields of the failure record. Therefore, if a new RAC file is added to the dataset and contains failure events with no corresponding classification rules, then the scripts are prepared to identify all these cases, support their manual classification, and incorporate the newly created classification rules into the filtering rules database [14]. Table 3.3, shows a summary of the whole failure dataset per group. Note that in terms of the number of failures, the most prevalent failure category is $USER_{APP}$, followed by OS_{SVC}, OS_{KNL}, and OS_{APP}.

Table 3.3. Failure characterization per group.

	G1	G2	G3	G4	Total
Sampling period (days)	331	592	378	1,408	2,709
Total of computers	259	262	37	106	644
Total of failures	2,955	21,490	2,299	15,465	42,209
% OS_{KNL}	31.03	1.76	1.52	1.38	3.66
% OS_{SVC}	3.82	12.40	9.31	8.13	10.07
% OS_{APP}	2.03	1.88	2.48	4.49	2.88
$\%USER_{APP}$	63.11	83.96	86.69	85.99	83.39
OS failures normalized by total of computers	11.41	82.02	62.14	145.90	65.54

Only in G1, Kernel failures predominated in relation to the failures of the other OS failure categories (OS_{SVC} and OS_{APP}). This may indicate the influence of environmental factors (e.g., computer configurations and workloads) present in this group, which contributed to these failures [30]. Note that the computers with the greatest number of failures in the four categories were concentrated in groups G2 and G4.

The data collecting and characterization of the field failure data presented so far were part of previous related work (e.g., [29], [56]) developed in our research group. Hence, the contribution of this thesis for this chapter follows.

To identify the main failed components in each category, all failures in the dataset were subdivided into 1,366 failure types; 35 belong to the OS_{KNL} category, 55 to OS_{SVC}, 23 to OS_{APP}, and 1,264 to $USER_{APP}$. I relied on the field *ProductName* to subdivide most failures. However, I also had to analyze the field *InsertionStrings* for most OS_{KNL} failures. For example, in Table 3.4 the five OS_{KNL} failure events have the same *SourceName* (MS-Windows-WER-SystemError Reporting) and *ProductName* (Windows), however, they have different hexadecimal values for the field *InsertionStrings*. These hexadecimal values, named *Stop Codes*, indicate the OS Kernel component that failed and the possible causes of the failure. The meaning of each *Stop Code* was obtained from the official OS technical documentation [73]. The column "OS Failure Type" contains the OS_{KNL} failed components obtained through the *Stop Codes*.

Investigating the OS Service category in detail, I found, based on the *SourceName* field of its failure records, that 41.81% (23 out of 55) of the failure types were related to software updates performed by the *Windows Update Service* (WUS), which is the standard software update mechanism in Win7; i.e., failures of this service while executing software update procedures. These WUS failures corresponded to 87.95% of the total amount of failures in the OS service category (3,737 out of 4,249). Given this prevalence of WUS-related failures in the OS Service category, they were grouped into 23 subtypes within the type WUS.

Figure 3.4 gives an overview of the data stratification performed in this study. I first stratified the failures into failure categories, and then into failure types. Specifically for the *Windows Update Service* (WUS), which targets different software components in the OS, I created subtypes to precisely identify which update routine failed while it was performing software update maintenance tasks.

Table 3.4. Types of OS Kernel failures.

InsertionStrings	SourceName	ProductName	OS Failure Type
0x00000116	MS-Windows-WER -SystemErrorReporting	Windows	VIDEO_TDR_ERROR
0x000000c2	MS-Windows-WER -SystemErrorReporting	Windows	STORAGE_DRIVER
0x00009088	MS-Windows-WER -SystemErrorReporting	Windows	BAD_POOL_CALLER
0x00000024	MS-Windows-WER -SystemErrorReporting	Windows	NTFS_FILE_SYSTEM
0x000000f4	MS-Windows-WER -SystemErrorReporting	Windows	CRITICAL_OBJECT_TERMINATION

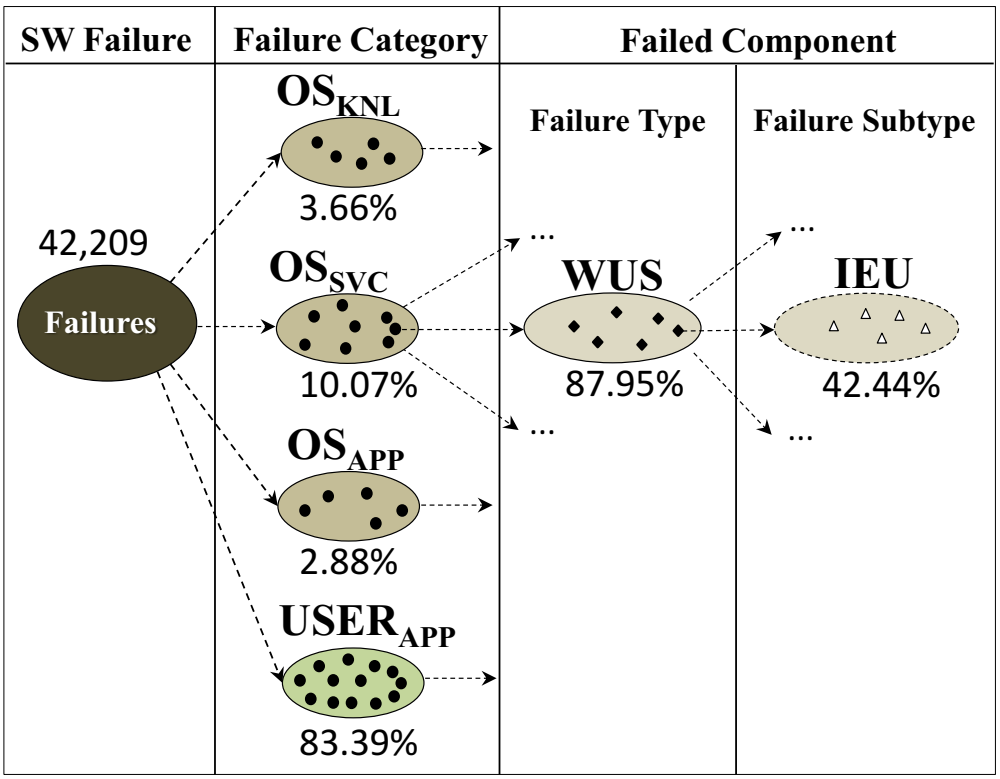


Figure 3.4. Dataset stratification.
Adapted: Dos Santos et al. (2018) [35]

Table 3.5 lists the five types of failures that consistently occurred in the dataset analyzed. They were ranked by three criteria; (i) greater number of groups the type is observed in; (ii) greater number of computers the type is observed in; (iii) greater number of occurrences of the type. The numbers in parentheses represent the total number of each element, e.g., the number in parentheses in column *Category* indicates that there are, in total, 4 different categories of failures in our dataset.

The most consistent failure in the dataset is the *iexplore*, which is a failure in the Internet Explorer browser. Other consistent $USER_{APP}$ failures are from productivity applications of Microsoft Office, Microsoft Word (*winword*), and Microsoft Excel (*excel*). The sole representative of the OS Application category was the *explorer.exe*. This application is responsible for two important features in the Win7 operating system: window and file management. The former implements the default desktop environment in Win7's graphical interface, while the latter is used to access and navigate the file system in the disk and network storage drives, at the file and directory levels. Both features are commonly labeled Windows shell in the Windows OS family's jargon. The most commonly observed type of Kernel failure was *StartupRepair*, which indicates that unspecified changes to the OS configuration occurred and it restarted to an earlier restore point [74].

I created three different work samples extracted from the above-described dataset — the first sample contains only OS failures (Sample_OS), the second contains only User Application failures (Sample_$USER_{APP}$), and the third encompasses the whole dataset, i.e., both OS and User Application failures (Sample_ALL).

Table 3.5. Consistent failure types.

Category (4)	Type [Subtype] (1,366)	Groups (4)	Computers (644)	Occurrence (42,209)	%
$USER_{APP}$	iexplore	4	329	6,495	15.39
OS_{APP}	explorer	4	205	586	1.39
$USER_{APP}$	winword	4	197	1,057	2.50
$USER_{APP}$	excel	4	160	874	2.07
OS_{KNL}	StartupRepair	4	160	237	0.56

4. METHOD

4.1 Introduction

In this chapter, I describe the method proposed to analyze and predict multiple-event failure occurrences. It is important to point out that all the described procedures are automated.

Section 4.2 describes my proposed method to discover patterns of failure associations. Along with this approach, a taxonomy designed to define several concepts used is also presented.

Section 4.3 describes the algorithm developed to discover patterns of failure sequences.

Section 4.4 describes the proposed prediction method based on patterns of failure sequences.

Note that I used the concepts of failure association (Section 4.2) and failure sequence (Sections 4.3 and 4.4) to determine the relationship patterns among the multiple-event failures investigated. Figure 4.1 compares both concepts by applying the methods described in sections 4.2 and 4.3 on the same failure log. Section 4.4 adopts a similar approach as presented in Section 4.3 to find patterns of failure sequences, which later are used in the proposed prediction method.

Observe that for Failure Association in Figure 4.1, the method identifies failures before and/or after, within a fixed search interval, from two occurrences of *Failure_C*. In this case, *Failure_C* is considered a reference to search for associations where the failures before and/or after should occur consistently with the reference. The references are failures that occurred in the greatest number of groups in the failure sample (i.e., at least three out of the four groups considered in this research). Moreover, observe that the failure associations have three types of configurations, reference with only a failure after (e.g., *Failure_C*→*Failure_H*), reference with failures before and after (e.g., *Failure_A*→*Failure_C*→*Failure_D*), and reference with only a failure before, which is not shown in Figure 4.1. Therefore, the associations have a maximum length (i.e., number of failures in the association) of three failures. Furthermore, the repetition of the same failure event is allowed in multiple associations, as can be seen for *Failure_A* in Figure 4.1, which is part of the associations *Failure_A*→*Failure_C*→*Failure_D* and *Failure_A*→*Failure_C*→*Failure_E*. Finally, the associations can be composed of consecutive failures (e.g., *Failure_C*→*Failure_H*) or not (e.g., there is a *Failure_B* between *Failure_A* and *Failure_C* that does not belong to any association because *Failure_B* is a random failure that does not occur consistently with *Failure_C* in other logs of the sample).

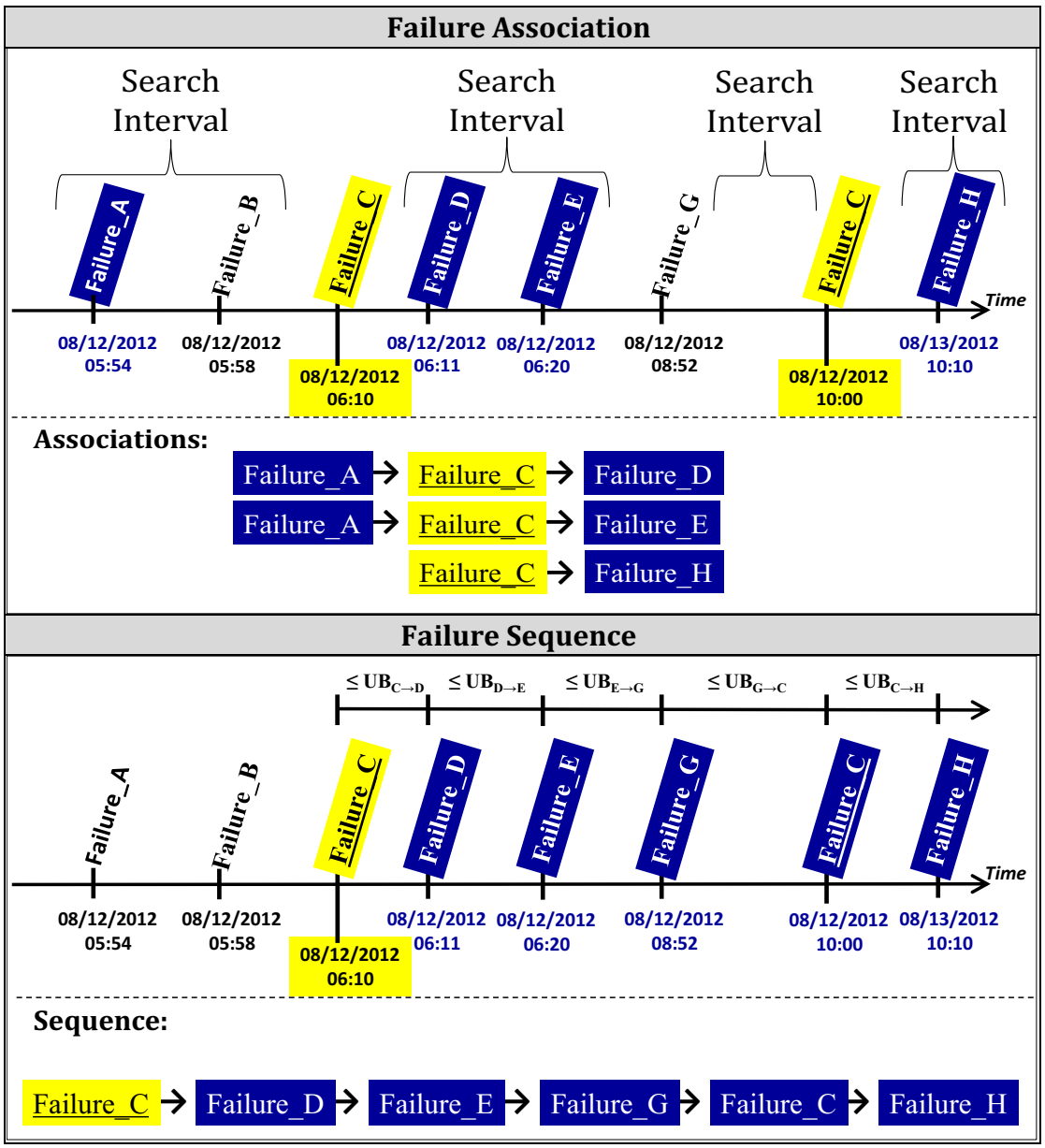


Figure 4.1. Failure association vs failure sequence.

In contrast, Failure Sequence refers to consecutive failures with no length limitation. Moreover, a failure event can only compose one sequence (i.e., event repetition is not allowed). To discover the sequences, the method checks if pairs of failures systematically occur in multiple computers and groups of computers. For example, the first pair of failures depicted in Figure 4.1 is *Failure_A→Failure_B*. If this pair is not observed in other groups of computers, it is not considered. Consequently, different from the first method, there is no relationship between *Failure_A* and *Failure_C* for this occurrence of multiple-event failures because of the random *Failure_B* between them. However, since *Failure_B* is just a random failure that occurred at that time, other occurrences of *Failure_A* happening immediately before *Failure_C* may be observed in the sample and the relationship may be considered. Furthermore, note that there is no fixed search interval in the Failure Sequence analysis. In this case, the method analyzes every pair of consecutive failures, and only the ones with Δt less or equal to the Upper Bound (UB) of the confidence interval for the median values of Δts are considered. The confidence interval is calculated by grouping the Δt of all occurrences of a pair in

the sample. For example, the method groups all occurrences of the pair *Failure_C→Failure_D* in the sample then calculates the confidence interval for the median values of their Δts, and subsequently, the Upper Bound ($UB_{C\rightarrow D}$) is also calculated. Finally, the method checks if the current occurrence been analyzed has a Δt less or equal to $UB_{C\rightarrow D}$. What stands out the most in this approach is the flexibility of using different time thresholds to determine the relationships of the failures. As can be seen in Figure 4.1, *Failure_G* was not considered in the first method because of the fixed time interval but was considered in the second, which uses a particular time threshold based on Upper Bounds for every pair of failures.

Important to highlight that my approach is not related to pattern recognition, but pattern discovery. Although it is not uncommon that both terms are used interchangeably, I emphasize that this method discovers patterns rather than recognizes them. The main difference is that in the former there is no prior knowledge of the patterns, as it is necessary for the latter; that is why this study is classified as an exploratory one.

The proposed method took advantage of the temporal order of the failure occurrences recorded as time series. According to [76], time-series data occur naturally in different application areas, such as economics, environmental modeling, and demographics, among others. Therefore, discovering patterns in time series is considered an important problem since it could help understand the phenomenon or problem, and even predict future behaviors. It is known from the literature that a time series is a set of ordered observations on a quantitative characteristic of an individual or collective phenomenon, taken at different points of time. Although theoretically, it is not essential, in practical terms most time series techniques assume that these data points are equidistant in time [77]. In the problem investigated in this study, the dataset is a series of OS failure records ordered in time; however, the observations are not equidistant in time.

The approach to analyzing unevenly spaced time series is to transform the data points into equally spaced observations, and then applying existing time-series methods for equally spaced data. However, according to the literature (e.g., [78], [79], [80], [81], and [82]), transforming data in such a way very often introduces significant and hard to quantify biases. In addition to that, since my analyses involve not a single type of OS failure, but multiple types of OS failures ordered in time, multivariate time series methods should be applied to evaluate possible failure associations and related properties. This could amplify the above-mentioned biases transformation problem, given that the multiple series I deal with in this study show not only unequal observations in time but also different numbers of observations spread to different computers. For these reasons, I decided to propose a new approach that takes into consideration all these specifics inherent to the investigated problem domain [30].

4.2 Failure Association Discovery Protocol

First, I investigate the existence of patterns of failure associations. For this purpose, I introduce a failure association discovery protocol that identifies failure associations exhibiting consistency across different computers used in the same as well as different workplaces. Initially, a taxonomy had to be established to define

several concepts used in the failure association discovery protocol itself. The following section contains a description of the elements of this taxonomy.

4.2.1 Taxonomy

- **Reference failure (RF)**: A failure that serves as a base reference to search for failure associations.
- **Previous failure event (PFE)**: Any failure event that occurs before an RF. The time gap between a PFE and an RF is called ΔtP.
- **Next failure event (NFE)**: Any failure event that occurs after an RF. The time gap between an RF and an NFE is called ΔtN.
- **Search Interval (SI)**: The time gap before and after an RF used to search for PFEs and NFEs.
- **Failure Association Candidate (FAC)**: A combination of failure events, found when PFEs and/or NFEs occur alongside an RF within a given SI. This protocol considers three FAC configurations:
    - **PFE→RF**: corresponds to a combination of events obtained from the analysis of failure events that occurred before a reference failure in a defined SI.
    - **RF→NFE**: corresponds to a combination of events obtained from the analysis of failure events that occurred after a reference failure in a defined SI.
    - **PFE→RF→NFE**: corresponds to a combination of events obtained from the analysis of failure events that occurred before and after a reference failure in a defined SI.

Note that in this representation of the three FAC configurations, the reference failures are always underlined and that the symbol "→" does not necessarily imply causation; this symbol indicates the temporal order of the failures belonging to the combination analyzed. Table 4.1 lists examples of all three considered FAC configurations.

Table 4.1. Examples of failure association candidates.

FAC Configuration	*ProductName*	*SourceName*	TimeGenerated
PFE→RF	dllhost.exe	Application Error	12/11/2012 09:29
	explorer.exe	Application Hang	12/11/2012 09:54
	svchost.exe	Application Error	12/11/2012 18:06
RF→NFE	rundll32.exe	Application Error	03/12/2012 06:59
	explorer.exe	Application Hang	03/12/2012 14:28
	dllhost.exe	Application Error	03/12/2012 14:46
PFE→RF→NFE	dllhost.exe	Application Error	08/06/2012 18:59
	explorer.exe	Application Hang	08/06/2012 19:03
	dllhost.exe	Application Error	08/06/2012 19:11

These examples in Table 4.1 are shown for didactical purposes only and are not part of the study results. Each example is a combination of failure events that occurred within a search interval of 2 h, before and/or after the reference failure. Through analysis of the RAC record field *TimeGenerated* (Section 3.3), which provides the date and time the failure was recorded, it is possible to identify the time order the failures occur. The shaded rows highlight the failure events that compose the FACs, while the rows with the underlined field *ProductName* indicate the respective reference failures (RFs). All reference failures listed in the table are related to the *explore.exe* program (OS Application).

For the FAC configuration PFE→RF, the combination of failure events is represented by *dllhost.exe*→*explorer.exe*. Thus, all *dllhost.exe* failures that occurred before the *explorer.exe* failures, within the 2 h search interval, shall be investigated. The same rationale applies to the other two configuration examples (RF→NFE and PFE→RF→NFE).

Despite the case shown in the third combination, in Table 4.1, having PFE equal to NFE, the occurrences in which both failures are different were also considered in this study, such as in *svchost.exe*→*explorer.exe*→*dllhost.exe*.

- **Failure Association (FA)**: An FA is detected when the same FAC repeats itself, systematically, in multiple computers from different groups.
- **Rankings**: To help identify an FA, a ranking is created with the FACs ordered by the number of occurrences in different computers and groups. The more a FAC repeats itself in different computers from different groups, the higher it will be ranked; this indicates that the candidate association has consistent evidence to be considered a failure association pattern.

In this study, rankings were created for each search interval and FAC configuration (e.g., ranking of the FAC configuration PFE→RF for the SI of 4 h). The composition of a ranking obeys the following criteria, ordered according to their importance for this study:

1. Greater number of groups the FAC is observed in.
2. Greater number of computers the FAC is observed in.
3. Greater number of occurrences of the FAC.
4. The smaller standard deviation of ΔtP and/or ΔtN for the FAC.

Therefore, firstly, FACs that appear in many different groups have more evidence to be considered association patterns. In case of a tie in one or more criteria, then the next criterion is evaluated following the order cited above. To apply these concepts above-mentioned, I propose a failure association discovery protocol that consists of seven major stages, which are described below.

4.2.2 Protocol

First Stage:

The first stage selects the reference failures (RFs) and defines the search intervals (SIs) of interest. To define the reference failures (RFs), the occurrence of all OS failure types found in the dataset, per group of computers, must be

examined; the more occurrences of the same failure type are observed in different groups, the greater its consistency and thus its importance for this study. Therefore, given that the dataset is composed of four groups of computers (G1, G2, G3, and G4), I defined that the RFs are failures whose occurrences are observed in at least three groups. In total, 28 reference failures with 6,351 occurrences were found, from which 56.24% appear in three groups and 43.76% appear in four groups. Tables A.1 and A.2, in the appendix, show the RFs found.

Regarding the search intervals (SIs), given that the investigated failures are from desktop computers, I adopted three intervals: 2 h, 4 h, and 8 h, which are supposedly close to the average working time of these desktop computers in their respective work environments.

Second Stage:

In this stage, the protocol identifies all previous failure events (PFEs) and/or next failure events (NFEs) for each RF, within the SIs of interest. First, it identifies an occurrence of the RF investigated, and based on the field *TimeGenerated* (*TG*), the ΔtP and ΔtN of the failure events that occurred before and after the RF, respectively, are calculated to determine which of these events occurred within a given SI, so that they can be added to the *lrpn* list.

The *lrpn* is a list containing all occurrences of the RF researched and its PFEs and/or NFEs that are within the investigated SI. Therefore, each RF has one *lrpn* per SI, e.g., RF = *Failure_A* has three *lrpn* lists: *lrpn- Failure_A*-2 h, *lrpn- Failure_A*-4 h, and *lrpn- Failure_A*-8 h.

Third Stage:

At this stage, the protocol searches for all PFEs and/or NFEs in all *lrpn* lists created in the previous stage, to identify the failure types that occurred along with a given RF in at least three groups, since it wants to find failure association patterns that consistently appear in multiple groups. The found failure types are added to a list of candidate failure events (*LCE*) per RF.

Table 4.2 shows an example of the failure events added to the *LCE* for the case where RF = *Failure_A*. Only failures of *Failure_A* and *Failure_B* are added to the *LCE-Failure_A* since they occurred in at least three groups. From the *LCE* list, all theoretical possibilities of failure association candidates are generated to be considered in the next three stages of the protocol, as illustrated in Table 4.3. Based on this table, one can see that the case where RF = *Failure_A* and *LCE-Failure_A* = {*Failure_A*, *Failure_B*} produced eight theoretically possible candidates of failures association.

Table 4.2. Groups with PFEs and/or NFEs for RF = Failure_A.

RF	PFE/NFE	Groups with PFE/NFE
Failure_A	*Failure_A*	G1, G2, G3, G4
	Failure_B	G1, G2, G4
	~~*Failure_C*~~	~~G2, G4~~

Table 4.3. Theoretical failure association candidates for RF = Failure_A and LCE-Failure_A = {Failure_A, Failure_B}.

FAC Configuration	FAC Event Combinations
PFE→RF	*Failure_A*→*Failure_A*
	Failure_B→*Failure_A*
RF→NFE	*Failure_A*→*Failure_A*
	Failure_A→*Failure_B*
PFE→RF→NFE	*Failure_A*→*Failure_A*→*Failure_A*
	Failure_A→*Failure_A*→*Failure_B*
	Failure_B→*Failure_A*→*Failure_A*
	Failure_B→*Failure_A*→*Failure_B*

Fourth Stage:

This stage consists of searching the dataset for failure event combinations that match the theoretical FACs generated in the previous stage, for the first configuration, i.e., PFE→RF. Considering the same example from Table 4.3, in this stage, the protocol would search the dataset for two possible (theoretical) FACs: *Failure_A*→*Failure_A* and *Failure_B*→*Failure_A*.

Figure 4.2 illustrates the execution of this protocol stage, based on the example described in the previous stage (Table 4.3), and considering the FAC event combination researched as *Failure_B*→*Failure_A*, i.e., PFE = *Failure_B* and RF = *Failure_A* within a search interval of 2 h. First, the protocol identifies an occurrence of the RF investigated; in Figure 4.2, the *Failure_A* that occurred at 13:10 is the first RF occurrence. Next, instead of verifying the PFEs, it verifies the failure events that occurred right after the RF = $Failure\_A^{13:10}$ (i.e., NFEs) that are within the 2 h SI. The only result found is NFE = $Failure\_B^{13:11}$. Since this NFE's failure type (*Failure_B*) belongs to the *LCE-Failure_A* (Table 4.3), this occurrence is discarded.

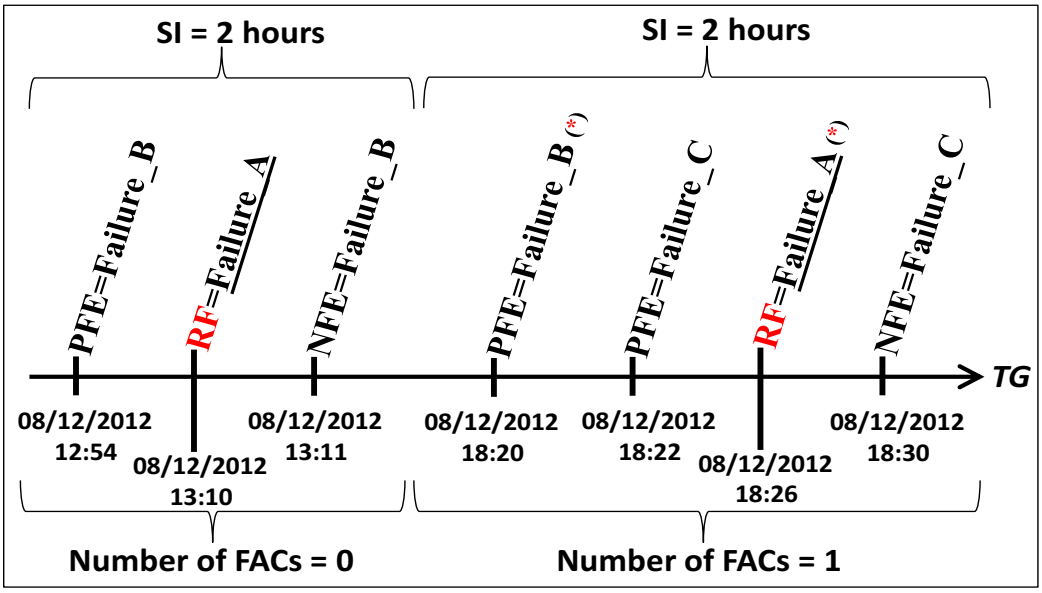


Figure 4.2. Searching for the FAC = Failure_B→Failure_A.

The first combination of failure is not considered because the protocol, in this stage, skips all cases in which the NFEs have the same types found in the *LCE*, given that these cases are already considered in the sixth stage of the protocol. For example, the first occurrence of RF at 13:10 comprises the combination of events represented by *Failure_B*→*Failure_A*→*Failure_B*, which is related to the FAC configuration PFE→RF→NFE that is analyzed in the sixth stage. Consequently, whenever the FAC configuration PFE→RF→NFE occurs, the other two configurations (PFE→RF and RF→NFE) do not occur.

Subsequently, once the protocol identifies the second occurrence of the RF ($Failure\_A^{18:26}$), it finds one occurrence of the FAC investigated, given that the found NFE does not have the same type present in *LCE-Failure_A*; thus, there are no possible occurrences of the FAC configuration PFE→RF→NFE and the protocol found a failure event with the same type of the PFE searched for the FAC investigated (*Failure_B*→*Failure_A*). In Figure 4.2, the combination of events that compose the FAC discovered ($Failure\_B^{18:20}$→$Failure\_A^{18:26}$) is marked with the symbol (*). All occurrences of the FAC are stored in the corresponding *lFAC* lists (i.e., Lists of Failure Association Candidates), which contains all occurrences of a FAC for a given SI.

Fifth and Sixth Stages:

These two stages of the protocol are similar to the fourth stage; focusing on finding the two remaining FACs configuration: RF→NFE and PFE→RF→NFE.

Seventh Stage:

In the seventh stage, the *lFAC* lists are analyzed. For each SI, the protocol counts the number of groups where it found the FAC occurrences and, based on the field *ComputerName*, the number of computers where these FACs appear. Subsequently, the number of PFEs and/or NFEs are accounted to identify the number of FACs occurrences. Finally, it calculates the standard deviations of ΔtPs and/or ΔtNs of all FACs discovered. Through the analysis of these quantities, the rankings of failure association candidates are created and ordered according to the four criteria presented in Section 4.2.1. Thus, the top-most entries in the rankings have the strongest evidence to be considered failure association patterns. Figure 4.3 gives an overview of all stages of the protocol.

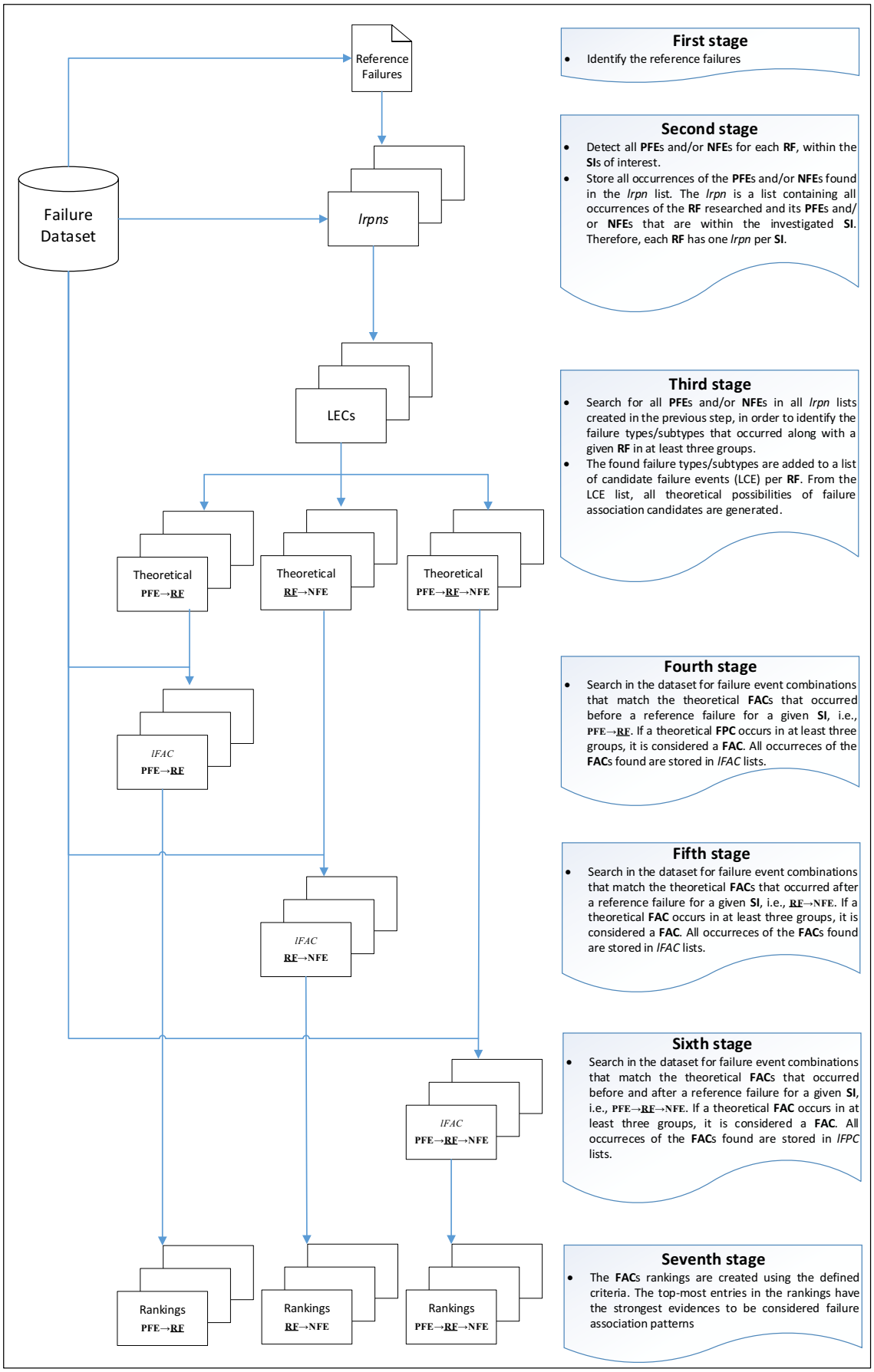


Figure 4.3. Protocol's execution flow.
Adapted: Dos Santos and Matias (2018)

4.3 Failure Sequence Discovery Protocol

The protocol described in Section 4.2 was designed to discover patterns of failure associations based on fixed search intervals. The adopted search intervals were defined in accordance with the average working times of the computers used in this research. However, to generalize the approach to any type of software system, I improved the previous method by developing the sequence discovery protocol presented in this section, whose objective is to find sequences of multiple-event failures under flexible search intervals. The protocol consists of five major stages:

First Stage:

For each computer failure log, the protocol finds its first failure and check if its next failure occurred on the same day or the next day. If so, the protocol stores the type and cause of its next failure and the Δt between them into two additional fields (*NextFailure* and Δt) created in the failure sample, for each failure event. Otherwise, the protocol moves to the next failure in the log and repeats the same procedure. Finally, it removes all failure records with no value in the two above-mentioned fields from the failure sample, i.e., all failure records with no subsequent failure events that occurred on the same day or the following day are removed [14].

- In the example presented in Figure 4.4 (1st stage), F1 is the first failure record in the log and has F2 as its next failure, which occurred on the same day as F1; the same occurs with F2 and F3, but in this case, F3 occurred on the following day. On the other hand, F3 does not have a subsequent failure, since F4 occurred two days apart; thus F3 does not contain a *NextFailure*, as well as the corresponding Δt. Since F4 is the last failure of the log, it also does not contain values assigned to fields *NextFailure* and Δt. Finally, the protocol removes both F3 and F4 from the failure sample, since they do not have subsequent failures. Note that the protocol is initially analyzing the association between two failures (i.e., pairs composed of a failure with its subsequent failure).

Second Stage:

Next, after checking for associations of failure pairs in all failure logs, the protocol filters the remaining sample to only include pairs that occurred in at least three out of the four groups of the sample (see Section 3.4), because the protocol was developed to identify recurrent and systematic associations between failure events [14].

- As shown in Figure 4.4 (2nd stage), the protocol groups all failure pair occurrences and check how many groups they were observed in. In the example, the pair F3→F4 was removed from the analysis because it was only observed in group G1.

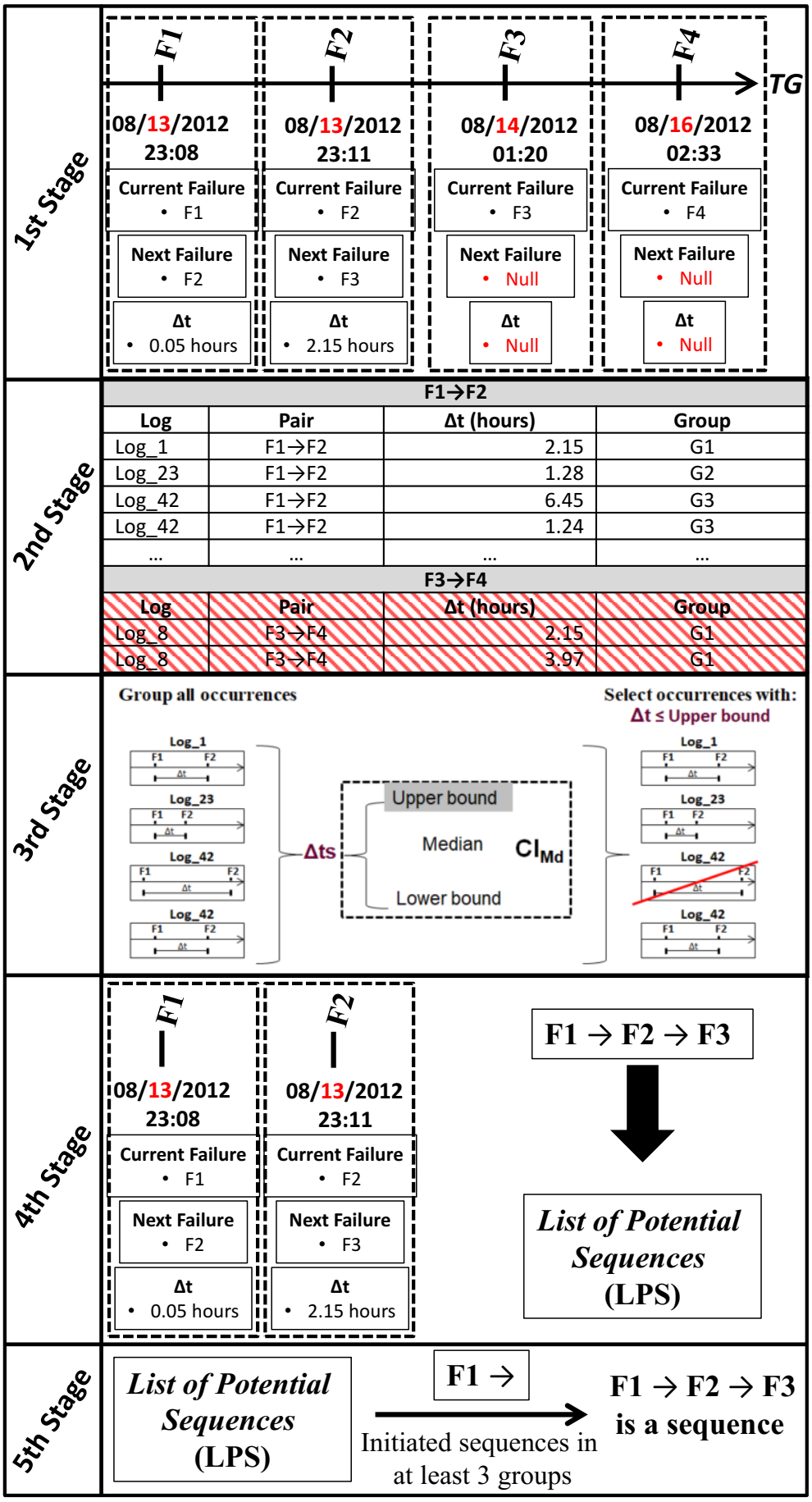


Figure 4.4. Sequence discovery protocol.

Third Stage:

The protocol includes failure pairs in which one failure occurred in one day and the other on the next day to account for failures occurring overnight (i.e., between 23:00 and 1:00). However, this can encompass outliers in the analysis. For example, if most occurrences of a failure pair have Δts less or equal to 2 hours, but in only one occurrence, the first failure occurred in the early morning (e.g., 03/20/2014 01:30) and its subsequent failure in the evening of the following day (e.g., 03/21/2014 23:45), the Δt between them would be of 46.25 hours, which is an outlier and must be disregarded. Thus, to avoid outliers and to guarantee a time consistency of the failure pairs, the protocol groups all occurrences of the same pairs of failures and computed the confidence interval around the median value of their Δts. The median was adopted instead of the mean because of the high variability of Δt values given the inclusion of overnight failures in the associations' time intervals. Next, the protocol computes the 95% confidence interval (CI) for the median values of Δt using the bootstrap resampling statistical approach, which according to [83] is an efficient, statistically sound, and accurate method to calculate confidence intervals for normal and non-normal datasets. Finally, the protocol removes all occurrences of failure pairs with Δt less or equal to the upper bound of the calculated confidence intervals [14].

- Figure 4.4 (3rd stage) shows this step for the pair F1→F2. Note that the protocol groups all occurrences of F1→F2 in the sample and calculate the confidence interval for the median (CI_{MD}) of the Δts of all the failure occurrences found. Later, the first occurrence of the pair F1→F2 in Log_42 is removed because its Δt (6.45 h) is greater than the calculated upper bound.

Fourth Stage:

The next stage is to discover possible sequences of failures in the remaining sample. For each computer log, the protocol identifies the first occurrence of a failure that has a subsequent failure (i.e., its field *NextFailure* is not Null) and sets it as an *OriginFailure*. Then, it checks if its next failure also has a subsequent failure, and so on, until there is no next failure. Following, the sequence found is added to a *List of Potential Sequences* (LPS) [14].

- In Figure 4.4 (4th stage) the first failure that has a subsequent failure is F1. Therefore, I set F1 as an *OriginFailure* and check if the next failures also have a subsequent failure until no subsequent failure is found. Consequently, the discovered sequence (i.e., F1→F2→F3) is stored in the LPS.

Fifth Stage:

In the last stage, the protocol filters the LPS to extract only sequences in which the first failure, to be called hereafter as *OriginFailure*, initiates sequences in at least three of the four groups of the dataset, given that the more occurrences of similar sequences are observed in different computers and groups, the greater their consistency and thus their importance for this study [14].

- As shown in Figure 4.4 (5th step), the protocol checks if F1 is the *OriginFailure* in potential sequences that occurred in at least three out of the four groups. If so, the potential sequence F1→F2→F3 is considered a sequence pattern.

4.4 Failure Prediction Approach

Based on the knowledge gained analyzing the outputs of the protocols described in sections 4.2 and 4.3, I developed an approach to predict failures based on patterns of previous failure events. As with any data-driven approach, it must contain the important requirement of efficiently process large datasets. Another important requisite to this approach is the ability to perform the pattern discovery and the failure probability computation steps automatically since my ultimate goal is to use this approach to implement online failure prediction. Considering the nature of the failure records found in real computer logs, in which most of the time the failure types must be handled as categorical variables, I found three promising methods to tackle this research problem; Multinomial Logistic Regression (w/ and w/o Ridge regularization), Decision Tree, and Random Forest.

I use patterns of failure sequences as input to the multinomial models. Following a similar rationale presented in Section 4.3, to find consistent sequences, the approach calculates the time between failures for each computer failure log using the steps shown in Figure 4.5.

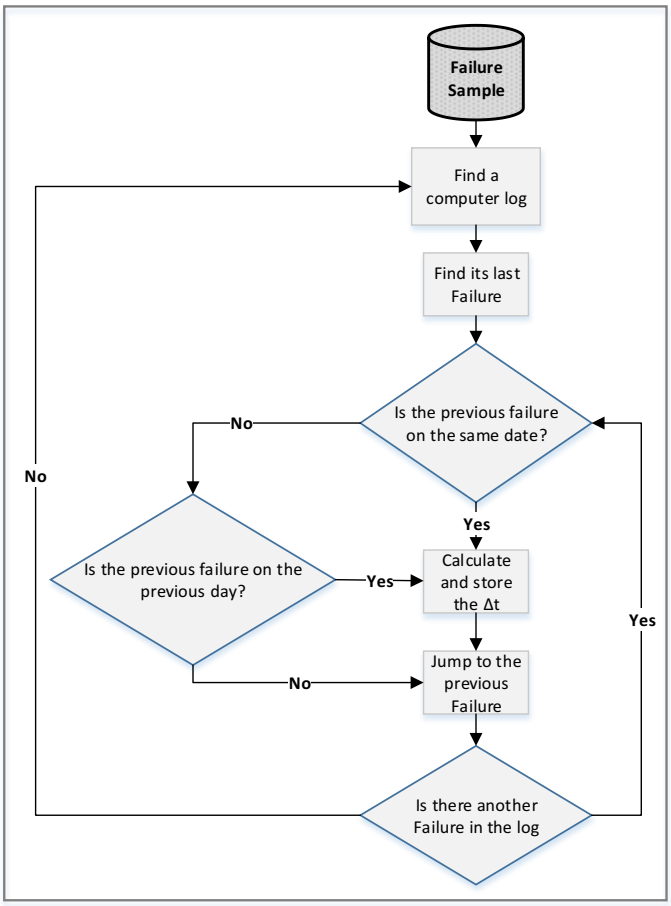


Figure 4.5. Filtering and collecting Δts between failures in computer logs.
Source: Dos Santos et al. (2020) [75]

Note that different from Section 4.3, the last failure of a sequence is the essential element of this analysis because it is the target event by the prediction analysis. Therefore, for each computer log in the sample, the approach finds its last failure record and then check if its previous failure occurred on the same day or the previous day. Next, the Δt between the last failure and next-to-last failure is computed. Finally, the category and type of the next-to-last failure and the computed Δt are stored into the three fields created for this purpose (*Previous_Fcategory*, *Previous_Ftype*, and Δt) in the last failure. Otherwise, the approach moves to the previous failure and repeats the same steps for its precedent failure. Finally, the approach removes all failure records with no value in the three above-mentioned fields from the failure sample, i.e., all failure records with no previous failure events that occurred on the same day or the previous day are removed [75].

Similar to Section 4.3, I avoid outliers by calculating the confidence intervals of all failure pairs found and selecting only those pairs in which their Δts are no longer than the upper bound value. For example, I found that the upper bound of the confidence intervals for the median value of Δts between OS_{APP} failures occurring before OS_{SVC} was 4.42 hours. Therefore, the approach checked if every occurrence of OS_{APP} before OS_{SVC} has a Δt less or equal to 4.42 hours, otherwise, they were removed from the work sample [75].

In addition, since I am interested in finding sequence patterns, the approach only considers sequences in which their last failure occurred as the last failure of sequences in at least three out of the four groups investigated in this study. Finally, I add these systematic sequences of failures as input to the prediction models, in which the first failures of the sequences (Previous Failures) represent the possible values for the independent variables and the last failures the possible values for the dependent variable. For example, suppose I discovered nine systematic sequences of two failures as shown in Figure 4.6. In this case, *Failure_D*, *Failure_E*, *Failure_F* are placed as the possible values of the independent variables (Previous Failures), and *Failure_A*, *Failure_B*, *Failure_C* are the possible values of the dependent variables (Last Failures). Finally, the prediction models calculate the probabilities for all combinations of values between the dependent and independent variables.

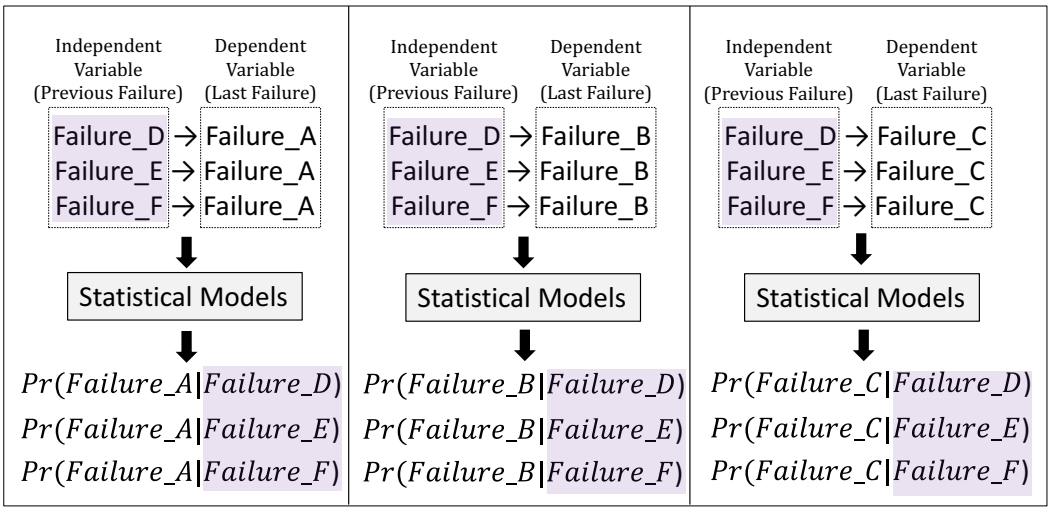


Figure 4.6. Input to the prediction models.

5. RESULTS

5.1 Introduction

This chapter presents the results of the empirical exploratory study conducted in this research. All methods explained in Chapter 4 are applied to the failure data described in Section 3.4.

Section 5.2 presents the results obtained discovering patterns of failure associations.

Section 5.3 presents the results obtained discovering patterns of failure sequences.

Section 5.4 presents the results obtained predicting failures based on patterns of failure sequences.

Figure 5.1 presents an overview of how the methods were applied to the failure data. The method to discover patterns of failure associations, which provided an initial analysis of multiple-event failures, was applied to only the OS failure sample (see Sample_OS in Section 3.4) in search for the three possible configurations of associations considered. The following methods (i.e., Failure sequence discovery protocol and Failure prediction approach), which are updated versions of the first, were applied to all three samples (see Sample_OS, Sample_$USER_{APP}$, and Sample_ALL in Section 3.4) studied in this research to provide a holistic view of multiple-event failures at the software layer.

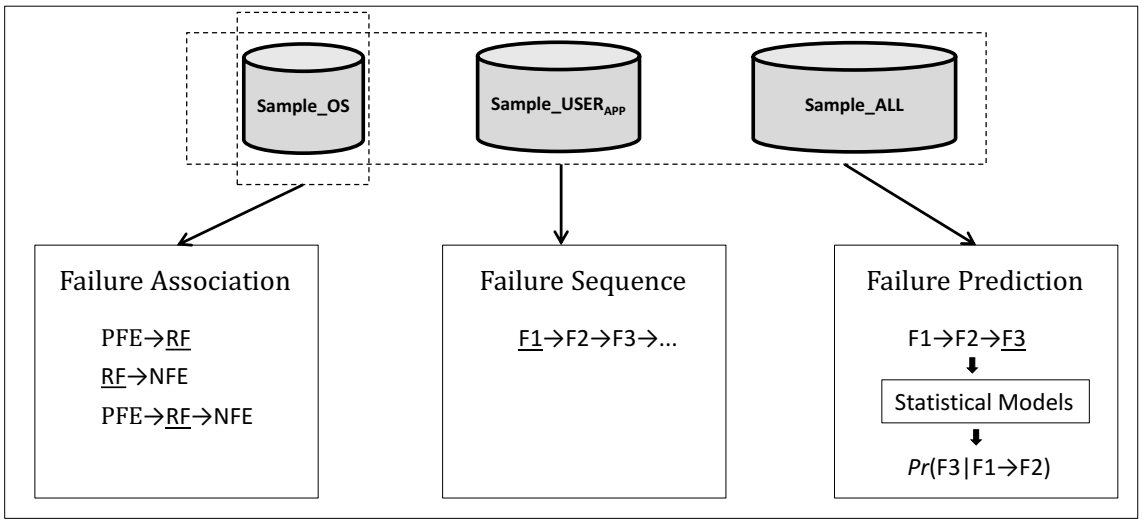


Figure 5.1. Overview of the methods and samples.

In the case of the Failure Sequence, the first failure (*OriginFailure*) is the essential element to discover failure sequences. On the other hand, the last failure is the essential element for Failure Prediction since its main concept is to calculate the probability of the last failure event to occur within a time interval upon the occurrence of a particular pattern of preceding failures (see Section 4.4).

5.2 OS Failure Associations

In this section, I present the results obtained applying the failure association discovery protocol to the Sample_OS. Table 5.1 lists the acronyms of OS failure types/WUS[subtypes] used in the rankings of failure association candidates (FACs) discussed in this section.

Tables 5.2, 5.3, and 5.4 present the top five FACs of all rankings. Each table shows three rankings according to the SIs investigated (2 h, 4 h, and 8 h). The column "Rank" contains the position of each FAC in the ranking.

It is noteworthy that the same combination of OS failure events can compose FACs in the three rankings of the same table, however, their positions may not be the same in all these rankings. For example, in Table 5.2, the FAC = *NetU→NetU* ranks third in both 8-hour and 4-hour rankings and also appears in the 2-hour ranking, but in the fourth position.

Table 5.1. OS failure types/subtypes acronyms used in the rankings.

Category	OS Failure types [Subtype]	Acronym
OS_{APP}	explorer.exe	EXP
OS_{SVC}	WUS [Windows Update]	WinU
OS_{SVC}	WUS [.Net Framework Update]	NetU
OS_{SVC}	WUS [Internet Explorer Update]	IEU
OS_{SVC}	WUS [Office Update]	OffU

Table 5.2. Top five entries in rankings of the configuration FAC = PFE→RF.

SI	Rank	PFE	RF	No. of Groups with the FAC	% of Computer with the FAC	No. of FAC Occurrences	SD
8 hours	**1**	EXP	EXP	4-4	6.3604	98	0.0913
	2	WinU	WinU	4-4	5.8304	554	0.0645
	3	NetU	NetU	4-3	2.4735	129	0.1139
	4	IEU	WinU	4-3	2.4735	28	0.0103
	5	OffU	OffU	4-3	2.1201	65	0.0542
4 hours	**1**	EXP	EXP	4-4	6.3604	(−) 81	0.0451
	2	WinU	WinU	4-4	5.8304	(−) 535	0.0504
	3	NetU	NetU	4-3	2.4735	(−) 111	0.0596
	4	IEU	WinU	4-3	2.4735	28	0.0103
	(^) 5	NetU	WinU	4-3	2.1201	55	0.0091
2 hours	**1**	EXP	EXP	4-4	(−) 6.1837	(−) 70	0.0257
	2	WinU	WinU	4-4	(−) 5.6537	(−) 436	0.0176
	(^) 3	IEU	WinU	4-3	2.4735	28	0.0103
	(ˇ) 4	NetU	NetU	4-3	(−) 2.2968	(−) 85	0.0251
	5	NetU	WinU	4-3	2.1201	55	0.0091

Table 5.3. Top five entries in rankings of the configuration FAC = RF→NFE.

SI	Rank	RF	NFE	No. of Groups with the FAC	% of Computer with the FAC	No. of FAC Occurrences	SD
8 hours	**1**	EXP	EXP	4-4	6.3604	95	0.0957
	2	WinU	WinU	4-4	6.0071	484	0.1057
	3	NetU	NetU	4-3	3.5336	151	0.1081
	4	WinU	IEU	4-3	3.0035	61	0.0852
	5	OffU	OffU	4-3	2.1201	82	0.0499
4 hours	**1**	EXP	EXP	4-4	6.3604	(−) 82	0.0460
	2	WinU	WinU	4-4	6.0071	(−) 398	0.0575
	3	NetU	NetU	4-3	(−) 3.3569	(−) 121	0.0602
	4	WinU	IEU	4-3	(−) 2.8269	(−) 56	0.0468
	5	OffU	OffU	4-3	(−) 1.9435	(−) 79	0.0062
2 hours	**1**	EXP	EXP	4-4	(−) 6.1837	(−) 70	0.0244
	2	WinU	WinU	4-4	(−) 5.8304	(−) 291	0.0205
	3	NetU	NetU	4-3	(−) 3.1802	(−) 95	0.0247
	4	WinU	IEU	4-3	2.8269	(−) 55	0.0093
	5	OffU	OffU	4-3	1.9435	79	0.0062

Table 5.4. Top five entries in rankings of the configuration FAC = PFE→RF→NFE.

SI	Rank	PFE	RF	NFE	No. of Groups with the FAC	% of Computer with the FAC	No. of FAC Occurrences	SD
8 hours	**1**	WinU	WinU	WinU	4-4	5.3004	23337	0.0354
	2	WinU	WinU	IEU	4-3	2.8269	6099	0.0502
	3	WinU	WinU	NetU	4-3	2.6502	12203	0.0371
	4	IEU	WinU	IEU	4-3	2.4735	860	0.0813
	5	IEU	WinU	WinU	4-3	2.2968	8622	0.0190
4 hours	**1**	WinU	WinU	WinU	4-4	5.3004	(−) 22702	0.0085
	2	WinU	WinU	IEU	4-3	2.8269	(−) 5247	0.0156
	3	WinU	WinU	NetU	4-3	2.6502	(−) 11873	0.0071
	4	IEU	WinU	IEU	4-3	2.4735	(−) 690	0.0247
	5	IEU	WinU	WinU	4-3	2.2968	(−) 8579	0.0061
2 hours	**1**	WinU	WinU	WinU	4-4	5.3004	(−) 22574	0.0021
	2	WinU	WinU	IEU	4-3	2.8269	(−) 5157	0.0032
	3	WinU	WinU	NetU	4-3	2.6502	(−) 11834	0.0006
	4	IEU	WinU	IEU	4-3	(−) 2.2968	(−) 640	0.0069
	5	IEU	WinU	WinU	4-3	(−) 2.1201	(−) 8537	0.0020

To highlight the FACs position change in the ranks, I compared the three rankings from top to bottom, i.e., compared the 8-hour ranking with the 4-hour ranking and the 4-hour ranking with the 2-hour ranking. To show the changes and their directions, the symbol (^) in the column "Rank" is added once the FAC increases in the ranking, and the symbol (ˇ) is added once it decreases. The symbol (−) is also added to the tables to identify the decrease in the number of occurrences of a given FAC, or the decrease in the number of computers that contains the FAC's occurrences. This decreasing behavior is observed when comparing the occurrences of the same FAC in a ranking with lower SI to another ranking with the same FAC configuration, but larger SI.

In Table 5.2, an example of the decreasing behavior is the combination of events represented by *EXP→EXP*, which ranks first in the 8-hour and 4-hour

rankings, but the number of occurrences of this FAC in the 8-hour ranking, namely 98 occurrences, is higher than in the 4-hour ranking (81 occurrences). The same decreasing pattern can be observed for this FAC when comparing the 4-hour and the 2-hour rankings.

I created three codes to account for the number of groups that contain the FAC (fifth column): 4-4, 4-3, and 3-3. The first and the second digits in this code represent the number of groups with the RF and the number of groups with the FAC, respectively. The columns "PFE", "RF", and "NFE" list the types/subtypes of failures that compose the FAC configuration researched. The column "SD" contains the standard deviation values of ΔtP and/or ΔtN per FAC. This SD value indicates how much variation occurred between the average temporal proximity of a given RF and its PFEs and/or NFEs. Therefore, this value should be as low as possible, which indicates a greater consistency of the OS failure events that compose the FAC.

It has been observed that the FACs that appear on the top of all PFE→RF (Table 5.2) and RF→NFE (Table 5.3) rankings, for the three search intervals, are related to failures in the program *explorer.exe*. I found these FACs in several computers of different groups repeating systematically. As mentioned in Section 2.1.2, the *explorer.exe* provides important and largely used OS functionalities, making it very ubiquitous. In [84], the use of household computers of 1,434 users was studied for 19 months and the author found that the *explorer.exe* was the only program executed by all users, ranking second in global execution time. Such extensive use could be an explanation for its leading position in the rankings of failure association patterns presented here.

In second place, all rankings in tables 5.2 and 5.3 show the *Windows Update* (WinU). Similar to the *explorer.exe* failures, its FACs of configurations PFE→RF and RF→NFE are composed of failures of the same type/subtype.

In Table 5.2, in the 8-hour and 4-hour rankings, the third place is the *.Net Framework Update*, which occurs both as PFE and RF. In the 2-hour search interval, this failure event descends to fourth place in the ranking due to its lower occurrence with respect to the number of computers in which it appears. Thus, for the 2-hour search interval in third place is the *Internet Explorer Update* as PFE and *Windows Update* as the RF. This combination of events also appears in fourth place in the search intervals of 8 h and 4 h. The fifth place for the 8-hour search interval is *Office Update* as both PFE and RF. For the 4-hour and 2-hour search intervals, the pattern changes to *.Net Framework Update* as PFE and *Windows Update* as the RF.

In Table 5.3, third place in all rankings always shows *.Net Framework Update* as RF and NFE. Similarly, fourth place in all rankings is the combination of *Windows Update* as RF and *Internet Explorer Update* as NFE. Fifth place in the rankings in this table shows *Office Update* as both the RF and NFE for all search intervals.

In Table 5.4, where the PFE is analyzed simultaneously with the NFE, the first event combination in all rankings, FAC = *WinU*→*WinU*→*WinU*, follows the same arrangement as the other first FACs in the other rankings, i.e., having the RF of the same type/subtype as PFE and NFE. However, this is not true for the other

positions in this table. Observe that the other association candidates have the same reference failure (*WinU*) and all PFEs and NFEs are also from the OS Service category and related to software updates. Therefore, they belong to the type *Windows Update Service* (WUS). Note that all top five failure association candidates maintained the same ranking positions for the different search intervals analyzed, making them strong candidates to be considered as genuine failure association patterns (FAs).

Comparing all rankings, note that the first association candidates in all of them are similar since in all cases the PFE and/or NFE are of the same type/subtype as the reference failure (RF). This behavior was also observed in other positions of the rankings, for the configurations PFE→RF and RF→NFE. This failure association pattern suggests two possible scenarios [30]:

- The first scenario is the case where the same OS part (application/ service/Kernel) failed multiple times, one execution at a time, through repeated executions. For example, the OS application *explorer.exe*, which runs as a single task, fails and must be restarted, where the new task fails subsequently.

- The second scenario is the simultaneous execution of multiple tasks of the same OS part, in which two or more tasks fail at the same time or approximately at the same time. For example, the OS service *svchost.exe* runs as multiple tasks and a few of them could fail within a close time interval.

The same failure events that composed the FACs in all rankings of Table 5.2 also composed the FACs in all rankings of Table 5.3. This is due to several occurrences of the same type/subtype of RF in which the search interval of each one comprises all the other occurrences, which makes them PFEs or NFEs.

For example, let us analyze the occurrence of two consecutive *explorer.exe* failures one minute apart, i.e., *explorer.exe*$^{12:54}$ and *explorer.exe*$^{12:55}$. For the 2-hour search interval, FAC = *explorer.exe*$^{12:54}$→*explorer.exe*$^{12:55}$ is found when the failure events occurring before the RF are researched (Table 5.2). On the other hand, when the failure events occurring after the RF (Table 5.3) are researched, then FAC = *explorer.exe*$^{12:54}$→*explorer.exe*$^{12:55}$ is also found. Therefore, the same combination of failures is detected for the two FAC configurations, which makes the composition of the FACs similar in both tables. This behavior explains why the number of FAC occurrences (256,434) is higher than the total number of OS failures (7,010) since a given failure can occur as RF a single time for each of the three possible FAC configurations, but it can occur as PFE and NFE countless times.

5.2.1 Results Quality

In this subsection, I discuss the soundness of the presented findings. To discover genuine failure association patterns, the consistency of the failure association candidates is a major requirement.

In this study, consistency is directly related to how often a given combination of failure events (a failure association candidate) repeats itself across different computers from different groups (workplaces). In addition to comparing the

similarity of different occurrences of the same failure events combination concerning the temporal order of the failures, I further evaluate their variability in terms of the time between their failures.

Hence, to obtain quality results, I defined that the rankings must be created only with FACs occurring in at least three groups (out of four), thus guaranteeing that each failure association candidate appears in at least 75% of the groups investigated in this study.

In addition to observing the occurrence of the same failure association candidate in multiple computer groups, another important factor to evaluate the quality of the results is the standard deviation (SD) computed on the ΔtP and/or ΔtN values, for each FAC.

Based on values from column "SD" in tables 5.2, 5.3, and 5.4, it can be seen that all failure association candidates discovered do have low SD values, although the first entries in all rankings do not contain the lowest values. The analysis of these values for all association candidates shows that the difference between the highest and lowest standard deviations, of each ranking, is noticeably small. This low variability in the times between the RF and its PFE and/or NFE, across the different groups of computers analyzed, indicates good consistency of the failure association candidates discovered [30].

Although the dataset shows failures in the OS Kernel category, mainly in group G1 with over 80% of its failures in that category, only two FACs belonging to the OS Kernel were identified when analyzing all event combinations in the rankings; one for configuration PFE→RF and the other for RF→NFE, both within the search interval of 8 h. These two configurations were composed of the same type of failure (DRIVER_POWER_STATE_FAILURE), i.e., PFE for the first configuration, NFE for the second, and the RF for both were failures of the type DRIVER_POWER_ STATE_FAILURE. This type of failure occurs when a device driver or an OS routine running at the Kernel level enters in an inconsistent or invalid energy state [73]. This finding corroborates results from previous studies, where OS Kernel failures were less frequent than user-level applications' failures (e.g., [27], [29], and [56]). A likely reason why there were not many association candidates for the OS Kernel category is the system reboot that very often occurs when most of these failures happen, which lowers the chances of further failures of this category within the same search interval.

Based on the above-mentioned analyses, I evaluated all 256,434 FACs discovered in the Sample_OS and selected the top five FACs in each ranking to classify as genuine failure association patterns (FAs). This selection resulted in the 45 patterns shown in tables 5.2, 5.3, and 5.4. These FAs appear systematically, 153,511 times, in the analyzed work sample. Table 5.5 shows the number of occurrences of the FAs for the SIs and pattern configurations investigated. Through the column "Total", it can be seen that the configuration PFE→RF→NFE has a greater number of occurrences. This finding indicates that most of the OS failures observed in this study occur following a specific temporal order, which is evidence of association patterns among these OS failures [30].

Table 5.5. Number of failure association pattern occurrences per configuration.

SI	PFE→RF	RF→NFE	PFE→RF→NFE	Total
8 h	874	873	51,121	52,868
4 h	810	736	49,091	50,637
2 h	674	590	48,742	50,006
Total	2,358	2,199	148,954	153,511

5.2.2 Effects of Changing the Search Intervals

This subsection discusses the behavior of the failure association candidates when changing the search intervals (SI) parameters.

It is intuitive to expect that the number of computers with the same FAC, as well as the number of occurrences of the same FAC, would remain constant, or would decrease in shorter search intervals than in longer search intervals [30]. The first case happens due to the nearness of the time occurrence of the RF with its PFEs and/or NFEs, i.e., the PFEs and/or NFEs within a smaller or equal to the smallest defined SI (SI = 2 h) with respect to the RF. Thus, even though the SI is reduced (8 h to 4 h, or 4 h to 2 h) the number of FAC occurrences remains constant.

Figure 5.2 shows an example of the above-mentioned scenario with the combination of events represented by *WinU*→*IEU*, where there is only one occurrence of the FAC. The time difference between the PFE = WinU$^{13:00}$ and RF = IEU$^{13:30}$ is 0.5 h. Therefore, this FAC is present in all SIs (8 h, 4 h, and 2 h).

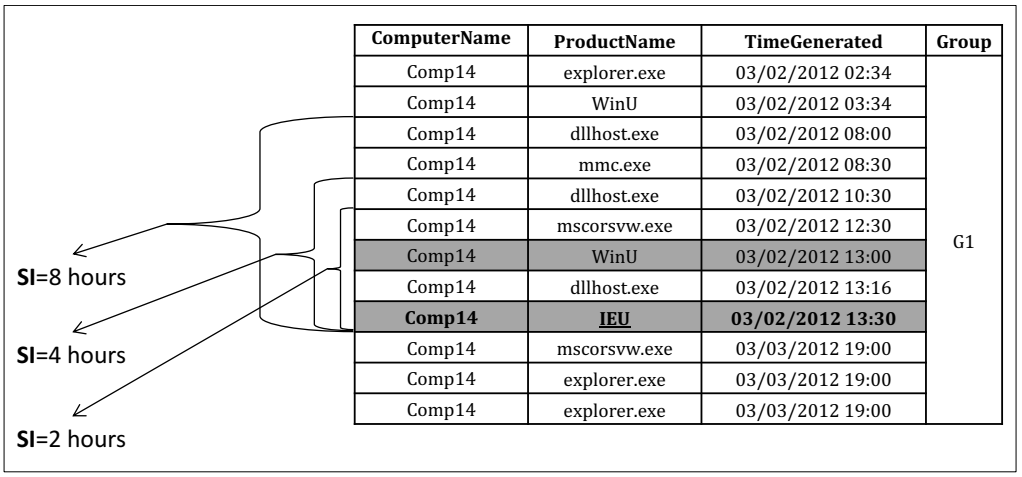
There is also the case of reducing the number of occurrences of a particular FAC, as well as the number of computers with these occurrences when the SI is decremented. This situation indicates that the time difference between the RF and its PFEs and/or NFEs is greater than the SI.

Figure 5.3 shows an example of this scenario, considering *WinU*→*IEU*. Note that the number of FACs changes according to the SI. For example, for SI = 8 h there are three occurrences, where the combination of events is represented by the following PFEs: WinU$^{08:00}$, WinU$^{10:30}$, and WinU$^{13:00}$; all with respect to RF = IEU$^{13:30}$. For SI = 4 h, the number of occurrences of this FAC decreases to 2, and the combination of events is represented by the PFEs WinU$^{10:30}$ and WinU$^{13:00}$, for RF = IEU$^{13:30}$. Lastly, for SI = 2 h there is only one FAC occurrence, which is *WinU*$^{13:00}$→*IEU*$^{13:30}$.

In addition to the two behaviors described above, I observed the increase of FACs and the number of computers containing these FACs occurrences when the SIs were decremented. This can only occur for the configurations PFE→RF and RF→NFE. This behavior was observed for the FACs with the configuration PFE→RF→NFE in the longest SIs; which prevents the occurrence of the other two configurations (PFE→RF and RF→NFE). This increase occurs when the SI is decremented (e.g., 8 h to 4 h) and a combination of events with the configuration PFE→RF→NFE, observed in the longest SI (8 h), does not occur in the reduced SI (4 h). Therefore, the combination of events with the configuration PFE→RF→NFE is

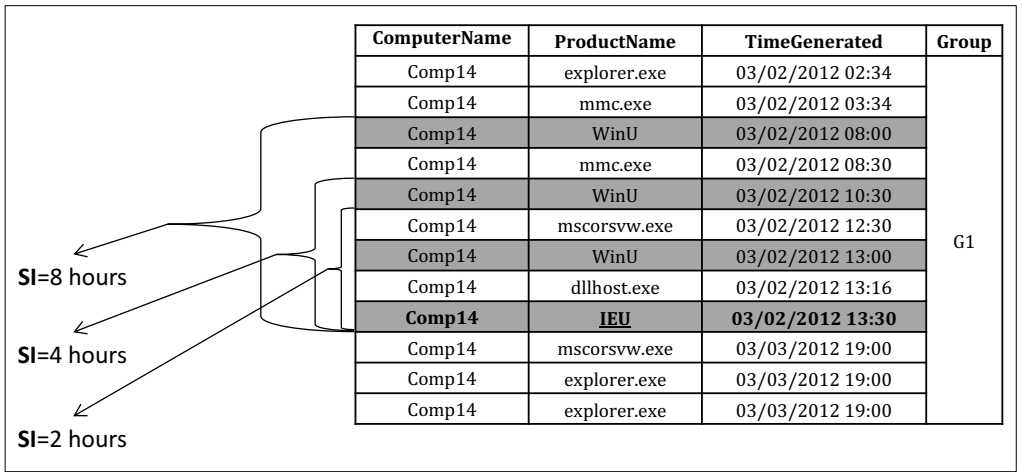
undone to the lower SI; consequently, it enables the discovery of combinations with the configuration PFE→RF or RF→NFE. As a result, there is an increase of FACs with configurations PFE→RF or RF→NFE, also causing an increase in the number of computers with these FACs. This behavior was observed twice in the results of this study for the configuration RF→NFE for a 4-hour SI (*IEU*→*IEU* and *IEU*→*WinU*).

Figure 5.4 shows an example of this scenario with the combination of events represented by *WinU*→*IEU*. Note that there is one occurrence of $WinU^{13:00}$→$IEU^{13:30}$→$WinU^{17:00}$, and according to the proposed protocol (Section 4.2.2), there is no occurrence of the FAC being investigated (*WinU*→*IEU*). However, for SI = 2 h, the FAC with the configuration PFE→RF→NFE is undone because the NFE = $WinU^{17:00}$ is not within this new SI being analyzed, composing one occurrence of $WinU^{13:00}$→$IEU^{13:30}$.



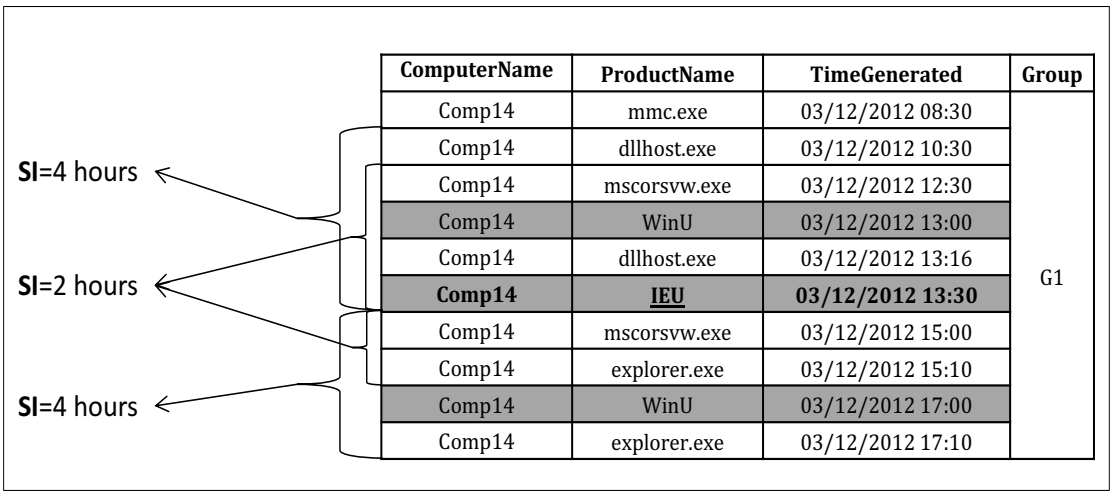
ComputerName	ProductName	TimeGenerated	Group
Comp14	explorer.exe	03/02/2012 02:34	G1
Comp14	WinU	03/02/2012 03:34	
Comp14	dllhost.exe	03/02/2012 08:00	
Comp14	mmc.exe	03/02/2012 08:30	
Comp14	dllhost.exe	03/02/2012 10:30	
Comp14	mscorsvw.exe	03/02/2012 12:30	
Comp14	WinU	03/02/2012 13:00	
Comp14	dllhost.exe	03/02/2012 13:16	
Comp14	**IEU**	**03/02/2012 13:30**	
Comp14	mscorsvw.exe	03/03/2012 19:00	
Comp14	explorer.exe	03/03/2012 19:00	
Comp14	explorer.exe	03/03/2012 19:00	

Figure 5.2. Number of FACs that remain constant by decreasing the SI.
Source: Dos Santos and Matias (2018) [30]



ComputerName	ProductName	TimeGenerated	Group
Comp14	explorer.exe	03/02/2012 02:34	G1
Comp14	mmc.exe	03/02/2012 03:34	
Comp14	WinU	03/02/2012 08:00	
Comp14	mmc.exe	03/02/2012 08:30	
Comp14	WinU	03/02/2012 10:30	
Comp14	mscorsvw.exe	03/02/2012 12:30	
Comp14	WinU	03/02/2012 13:00	
Comp14	dllhost.exe	03/02/2012 13:16	
Comp14	**IEU**	**03/02/2012 13:30**	
Comp14	mscorsvw.exe	03/03/2012 19:00	
Comp14	explorer.exe	03/03/2012 19:00	
Comp14	explorer.exe	03/03/2012 19:00	

Figure 5.3. Number of FACs that is reduced by decreasing the SI.
Source: Dos Santos and Matias (2018) [30]



ComputerName	ProductName	TimeGenerated	Group
Comp14	mmc.exe	03/12/2012 08:30	G1
Comp14	dllhost.exe	03/12/2012 10:30	
Comp14	mscorsvw.exe	03/12/2012 12:30	
Comp14	WinU	03/12/2012 13:00	
Comp14	dllhost.exe	03/12/2012 13:16	
Comp14	**IEU**	**03/12/2012 13:30**	
Comp14	mscorsvw.exe	03/12/2012 15:00	
Comp14	explorer.exe	03/12/2012 15:10	
Comp14	WinU	03/12/2012 17:00	
Comp14	explorer.exe	03/12/2012 17:10	

Figure 5.4. Number of FACs that is increased by decreasing the SI.
Source: Dos Santos and Matias (2018) [30]

5.3 Failure Sequences

Based on the association patterns discovered and discussed in Section 5.2, it can be seen that the independence assumption is not met for the software failure events analyzed in the OS work sample (Sample_OS). Moreover, the findings indicate that most OS failures observed occur following a specific temporal order, which raises the following research questions:

Do these OS failures occur in a sequential pattern? If so, do the observed patterns reflect a causal relationship between the failures? To shed some light on these questions, I searched for patterns of failure sequences in the OS failure sample (Sample_OS). Furthermore, I extended the analysis to include User Application failures (Sample_$USER_{APP}$), and I also analyzed the whole dataset, which encompasses both $USER_{APP}$ and OS failures (Sample_ALL).

Hereafter, I adopt the following notation to represent a sequence type: $Failure\_A^{Cause\_V} \rightarrow Failure\_B^{Cause\_W}$, which means failures of type B occurred after failures of type A. The superscripts indicate the failure cause and usually are shown as hexadecimal codes that are platform-specific. In the samples analyzed in this study, I found hundreds of failure sequences that met the criteria discussed in Section 4.3. To group them and make a more general analysis, I differentiated the failure sequence types by four characteristics:

- Failure types that compose the sequences

e.g., $Failure\_A^{Cause\_V} \rightarrow Failure\_B^{Cause\_W} \neq Failure\_C^{Cause\_V} \rightarrow Failure\_B^{Cause\_W}$

- The sequence order of failure types

e.g., $Failure\_A^{Cause\_V} \rightarrow Failure\_B^{Cause\_W} \neq Failure\_B^{Cause\_V} \rightarrow Failure\_A^{Cause\_W}$

- Length of the sequences

e.g., $Failure\_A^{Cause\_V} \rightarrow Failure\_B^{Cause\_W} \neq Failure\_A^{VCause\_V} \rightarrow Failure\_B^{Cause\_W} \rightarrow Failure\_C^{Cause\_Z}$

- Cause of failures that compose the sequences

e.g., $Failure\_A^{Cause\_V} \rightarrow Failure\_B^{Cause\_W} \neq Failure\_A^{Cause\_V} \rightarrow Failure\_B^{Cause\_Z}$

The information that identifies the failure causes was found parsing the RAC fields *Message* and *InsertionStrings*. For software update failures (WUS subtypes), this information was obtained through the *Error Codes*. These codes are hexadecimal values present in the field *Message* of each WUS failure record. For OS_{SVC} failures not related to software updates, $OS_{APP,}$ and $USER_{APP}$ failures, I analyzed the *Exception Codes* to obtain their causes. The *Exception Codes* were also found in the fields *Message* and *InsertionStrings*. However, the *Exception Codes* are not available in all failure records; I observed cases in which the system could not identify the causes of failure and other cases in which the component failed due to a hang. Therefore, I assigned "unknown" to the failure causes of the first case and "Hang" to the second. I did not perform this analysis for OS_{KNL} failures since their types, obtained through the *Stop Codes* (see Section 3.4), already identify their causes.

5.3.1 Sequence Types

Since I am analyzing 3 different samples, I found, as a result: 165 different OS failures sequences with 806 occurrences; 480 User Application failures sequences with 1,718 occurrences; and 640 sequences containing both OS and User Application failures with a total of 2,509 occurrences.

For the first sample (i.e., Only OS failures), all sequences found were composed of OS failures from the same category, in which most sequences (91.56%) consisted of OS_{SVC} failures, followed by OS_{APP} (7.07%) and OS_{KNL} (1.36%). The second sample contains only User Application failures. Finally, for the third sample, most sequences encompassed failures from the same category, but I also observed few occurrences of USER Application failures with OS failures. The majority of sequences consisted of User Application failures (66.56%), followed by {OS_{SVC}} (29.10%), {OS_{APP}} (2.07%), {$USER_{APP}$, OS_{APP}} (0.99%), {$USER_{APP}$, OS_{KNL}} (0.52%), {$USER_{APP}$, OS_{SVC}} (0.40%), and {OS_{KNL}} (0.36%). Note that sequences composed of OS_{KNL} failures are barely observed because these failures usually cause system reboots decreasing the chances of other failures nearby.

Since this study is focused on finding systematic and consistent patterns, I ranked the failure sequences by two criteria; (i) greater number of computers the sequence is observed in; and (ii) greater number of occurrences of the sequence. Table 5.6 presents the ranking result for each sample considered. The numbers in parentheses in the column labels represent the total number of each element, e.g., for the OS failures sample, the total number of computers that contain the sequences is 128 and the total number of sequence occurrences is 806. In general, I observed that, besides been consisted of failures from the same category, most sequences are composed of failures of the same type. Noticeably, the failures in both the application (*iexplore*) and update (*IEU*) of the Internet Explorer browser are predominant in the sequences.

Table 5.6. Most consistent failure sequences.

Sample_OS			
Sequence	**# of Computers (128)**	**Occurrence (806)**	**%**
Explorer^{Hang}→Explorer^{Hang}	18	30	3.72
IEU^{80242016}→IEU^{80242016}	14	14	1.74
IEU^{80242016}→WinU^{80242016}→IEU^{80242016}	8	8	0.99
WinU^{800f0902}→WinU^{800f0902}	6	6	0.74
IEU^{80070643}→IEU^{80070643}	5	316	39.21
Sample_USER$_{APP}$			
Sequence	**# of Computers (276)**	**Occurrence (1,718)**	**%**
iexplore^{0xc0000005}→iexplore^{0xc0000005}	77	215	12.51
iexplorer^{Hang}→iexplorer^{Hang}	77	152	8.85
winword^{hang}→winword^{hang}	29	47	2.74
iexplore^{0xc0000005}→iexplore^{0xc0000005}→iexplore^{0xc0000005}	28	43	2.50
iexplore^{0xc0000005}→iexplore^{0xc000041d}	25	43	2.50
Sample_ALL			
Sequence	**# of Computers (319)**	**Occurrence (2,509)**	**%**
iexplore^{0xc0000005}→iexplore^{0xc0000005}	78	219	8.73
iexplorer^{Hang}→iexplorer^{Hang}	75	149	5.94
winword^{hang}→winword^{hang}	30	48	1.91
iexplore^{0xc0000005}→iexplore^{0xc0000005}→iexplore^{0xc0000005}	28	41	1.63
iexplore^{0xc0000005}→iexplore^{0xc000041d}	25	41	1.63

For the Sample_OS in Table 5.6, the highest-ranked sequence consists of *explorer.exe* failures from the OS$_{APP}$ category, which is the only non-update OS failures observed in the rank. The *explorer.exe* is an OS program responsible for the GUI window and file navigation management. All other sequences are composed of WUS failures, specifically *IEU* and *WinU*. Note that the sequence *IEU*^{80070643}→ *IEU*^{80070643} is observed in five computers but represented almost 40% of the sequence occurrences found. For the entire first sample, around 46% of the failures that composed the sequences were *Internet Explorer Update* failures, followed by *Windows Update* (23.83%), *.Net Framework Update*(9.63%), *Visual Studio Update* (5.61%), and *Office Update* (5.08%).

Regarding the second sample (Sample_USER$_{APP}$), the majority of the sequences consisted of *iexplore* failure associations. The only sequence that did not encompass *iexplore* failures in the rank was composed of associations of Microsoft Word failures (*winword*), which ranked third. Most sequences of the second sample are composed of *iexplore* failures (62.56%), followed by *winword* (9.38%), *excel* (7.90%), *acrord32* (6.45%), and *chrome* (4.22%).

Since USER Application failures are abundant, the rank of the third sample (Sample_ALL), which included both OS and User Application failures, is the same as the second sample (Sample_USER$_{APP}$). However, note that some occurrence values from the rank of the second sample are different from the third.

Table 5.7. Most frequent causes in the failure sequences.

Sample_OS			
Cause	**Occurrence (2,673)**	**%**	**Description**
80070643	1,307	48.90	Indicates a failure during the software update installation process, whose most common cause is the malfunctioning of the .Net Framework installed on the system [85].
800706ba	429	16.05	Indicates a failure in the WUS (*Windows Update Service*) caused by the unavailability of the RPC (remote procedure call) server [86].
800b0100	181	6.77	This indicates that the files needed for the WUS to update the software were missing or corrupted [87].
c8000710	103	3.85	This indicates that the failure occurred due to the unavailability of disk space during the update installation process [88].
hang	95	3.55	Hang.
Sample_USER$_{APP}$			
Cause	**Occurrence (4,648)**	**%**	**Description**
0xc0000005	2,124	45.70	"STATUS_ACCESS_VIOLATION", The instruction at 0x%08lx referenced memory at 0x%08lx. The memory could not be %s [89]
hang	1,437	30.92	Hang.
0xc0000374	285	6.13	"STATUS_HEAP_CORRUPTION", A heap has been corrupted [89].
0xc0000096	259	5.57	"STATUS_PRIVILEGED_INSTRUCTION", {EXCEPTION} Privileged instruction [89].
0xc000041d	104	2.24	An unhandled exception was encountered during a user callback [90].
Sample_ALL			
Cause	**Occurrence (7,180)**	**%**	**Description**
0xc0000005	2,146	29.89	Already described in the table.
hang	1,493	20.79	
80070643	1,245	17.34	
800706ba	428	5.96	
0xc0000374	320	4.46	

Table 5.7 lists the most frequently observed causes of failures that compose the sequences found for the three samples analyzed. Note that the *Occurrence* and *%* columns regard the number of failures with each cause in the sequences found.

The first four causes of failures in the OS sample indicate that external factors (e.g., malfunctioning of the .Net Framework and unavailability of the RPC server) upon which the *Windows Update Service* (WUS) depends, generated failures on its normal execution. Thus, preventive management of these factors is a possible action to improve the reliability of this OS Service. Furthermore, the causes 80242016 and 800f0902, which are the cause of some of the most consistent sequences within the Sample_OS (see Table 5.6), indicate that the state of the update after its post-reboot operation has completed is unexpected [91] and the software updating routine could not be completed because Win7 was performing another servicing operation [92], respectively.

As for the User Applications sample, most causes of failures are related to problems related to memory management (i.e., 0xc0000005 and 0xc0000374). Also, there is a high incidence of failures due to hangs, which I could not obtain the specific cause. Furthermore, I collected and analyzed the *Faulting Modules* of all USER Applications failures, based on the fields *Message* and *InsertionStrings* of the event records, and observed that the *ntdll.dll* library that manages allocations inside large memory areas [73] is the most frequent module (14.05%) to fail in the sample, followed by *mshtml.dll* (13.36%), which is responsible for loading, parsing, and displaying HTML content in the Internet Explorer browser [93], and Unknown (6.38%).

Finally, for the third sample (Sample_ALL), the most common causes have already been described in Table 5.7 for the other samples.

It is intuitive to expect that the number of the same sequence occurrences would remain constant, or would decrease in the third sample given that this sample was created by adding OS failures to the second sample, which could not interfere with User Application sequences when OS failures occurred on a different timestamp from $USER_{APP}$ failures or could prevent User Application sequences from happening when OS failures occurred between $USER_{APP}$ failures, lowering the number of $USER_{APP}$ sequences. But I also observed the increase of sequence occurrences, for example, the sequence $winword^{hang} \rightarrow winword^{hang}$ occurred 47 times in the second sample and 48 in the third (see Table 5.6).

This increase behavior occurred when longer sequences observed in the second sample (e.g., $winword^{hang} \rightarrow winword^{hang} \rightarrow iexplore^{hang}$) do not occur in the third sample, but parts of the longer sequence compose smaller sequences in the third sample (e.g., $winword^{hang} \rightarrow winword^{hang}$). This happened because the inclusion of OS failures in the second sample changed the associations and Δts observed in the second sample. Regarding only the number of occurrences, the most frequent failures observed in sequences in the third sample are *iexplore* failures (40.21%), followed by *IEU* (16.62%), *WinU* (8.64%), *winword* (5.61%), and *excel* (5.08%).

5.3.2 Sequence Classes

Looking at the patterns shown in Table 5.6, there is a predominance of sequences composed of failures of the same type. Therefore, I further investigated the relationship between the failures in sequences by dividing the sequences into classes. The Sequence Type Class refers to the types/subtypes that compose the sequences, having three possibilities:

- **Homogeneous**: Sequences where all of the subsequent failures have the same type of the *OriginFailure* (see Section 4.3).
- **Entirely Heterogeneous**: Sequences where all of the subsequent failures have a different type of the *OriginFailure*.
- **Partially Heterogeneous**: Sequences where at least one subsequent failure has the same type of the *OriginFailure*.

I also classified sequences based on their causes (Sequence Cause Class):

- **Same**: Sequences where all of the subsequent failures have the same cause of the *OriginFailure*.

- **Entirely Different**: Sequences where all of the subsequent failures have a different cause of the *OriginFailure*.

- **Partially Different**: Sequences where at least one subsequent failure has the same cause of the *OriginFailure*.

Table 5.8 presents the frequency of sequences belonging to all combinations of classes considered above, based on every sample analyzed. Note that failures with the same type/subtype and cause are predominant in all samples, which indicate recurrent failure patterns formed by these sequences; this is useful information regarding the occurrence dynamics of these failures in the sequences, being valuable for the prediction models. Moreover, most sequences present common causes that provoke multiple failures over time (see Table 5.7), with no evidence of a causal relationship among the failures in the sequences. From a practical perspective, this finding indicates that fixing the common problem (e.g., missing files) would prevent new failures in the sequence.

Table 5.8. Sequence classes.

		OS		$USER_{APP}$		All	
Seq. Type Class	**Seq. Cause Class**	**Freq. (806)**	**%**	**Freq. (1,718)**	**%**	**Freq. (2,509)**	**%**
Homogeneous	Same	704	87.34	1,102	64.14	1,784	71.10
Homogeneous	Entirely Different	17	2.11	226	13.15	236	9.41
Homogeneous	Partially Different	1	0.12	120	6.98	120	4.78
Entirely Heterogeneous	Same	16	1.99	75	4.37	94	3.75
Entirely Heterogeneous	Entirely Different	8	0.99	127	7.39	143	5.70
Entirely Heterogeneous	Partially Different	-	-	4	0.23	4	0.16
Partially Heterogeneous	Same	42	5.21	19	1.11	59	2.35
Partially Heterogeneous	Entirely Different	3	0.37	10	0.58	16	0.64
Partially Heterogeneous	Partially Different	15	1.86	35	2.04	53	2.11

5.3.3 Sequence Lengths

Next, I analyzed the length of the sequences, which is the number of failures that compose them. Table 5.9 shows the five most observed lengths in each sample. Note that sequences composed of two failures, which is the minimum length of a sequence, are the most frequent in any sample. This is another valuable piece of information for the prediction models since we can conclude that given the predominance of this pattern the prediction should be focused on shorter sequence lengths. In general, we can see a clear negative correlation between the sequence lengths and the number of sequence occurrences, which is confirmed by the Spearman correlation coefficients calculated for each sample $s_{(OS)}$ = -0.85, $s_{(USERAPP)}$ = -0.96, and $s_{(ALL)}$ = -0.88. Therefore, the longer the sequence, the rarer they were observed in the samples; the longest sequence found has 58 OS failures and occurred once.

Table 5.9. Ten most frequent sequence lengths.

Sample_OS			
Length	**Frequency (806)**	**%**	**Total**
2	497	61.66	93.17%
3	154	19.11	
4	60	7.44	
5	30	3.72	
6	10	1.24	
Sample_USER$_{APP}$			
Length	**Frequency (1,718)**	**%**	**Total**
2	1,149	66.88	97.62%
3	301	17.52	
4	145	8.44	
5	58	3.38	
6	24	1.40	
Sample_ALL			
Length	**Frequency (2,509)**	**%**	**Total**
2	1,651	65.80	96.45
3	456	18.17	
4	198	7.89	
5	82	3.27	
6	33	1.32	

5.3.4 Sequence Durations

Another important property of the failure sequences is their time duration, which is the summation of all times between failures, Δts, in a sequence. First, we analyzed the median duration of sequences in each sample.

I observed that 2.81 hours (95% CI of 2.54 to 2.88) was the median duration of sequences in the Sample_OS; 0.043 hours (95% CI of 0.038 to 0.049) in the Sample_USER$_{APP}$; and 0.089 hours (95% CI of 0.072 to 0.108) in the Sample_ALL. Figure 5.5 presents histograms of sequence durations for each sample. The logarithmic scale $ln(x+1)$ is used to display the wide range of duration values since I found that some sequences have durations equal to zero hours, which occur when the Δts are less than one second since seconds is the lowest time unit recorded by the RAC instrumentation.

I noted that most sequences from the OS sample (Sample_OS) last for a time between 0 and 0.22 hours (approximately 13 minutes), followed by sequences with a duration between 2.32 to 3.06 hours. The durations of USER$_{APP}$ sequences are concentrated between 0 and 0.11 hours (approximately 6 minutes). Therefore, I investigate in more detail the sequences within the intervals 0 to 0.22 hours and 2.32 to 3.06 hours in the Sample_OS and the sequences within the interval 0 to 0.11 hours in the Sample_USER$_{APP}$.

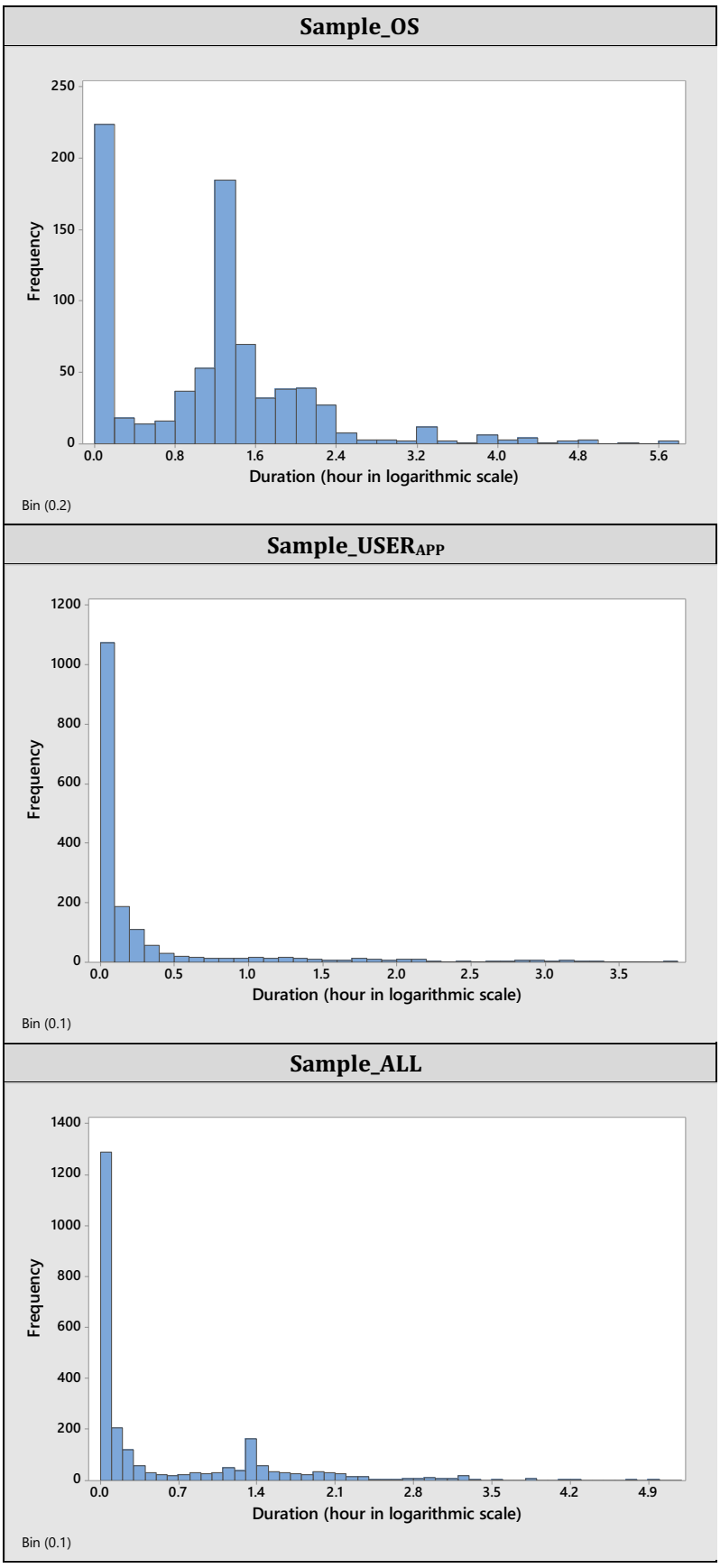


Figure 5.5. Histogram of failure sequence durations.

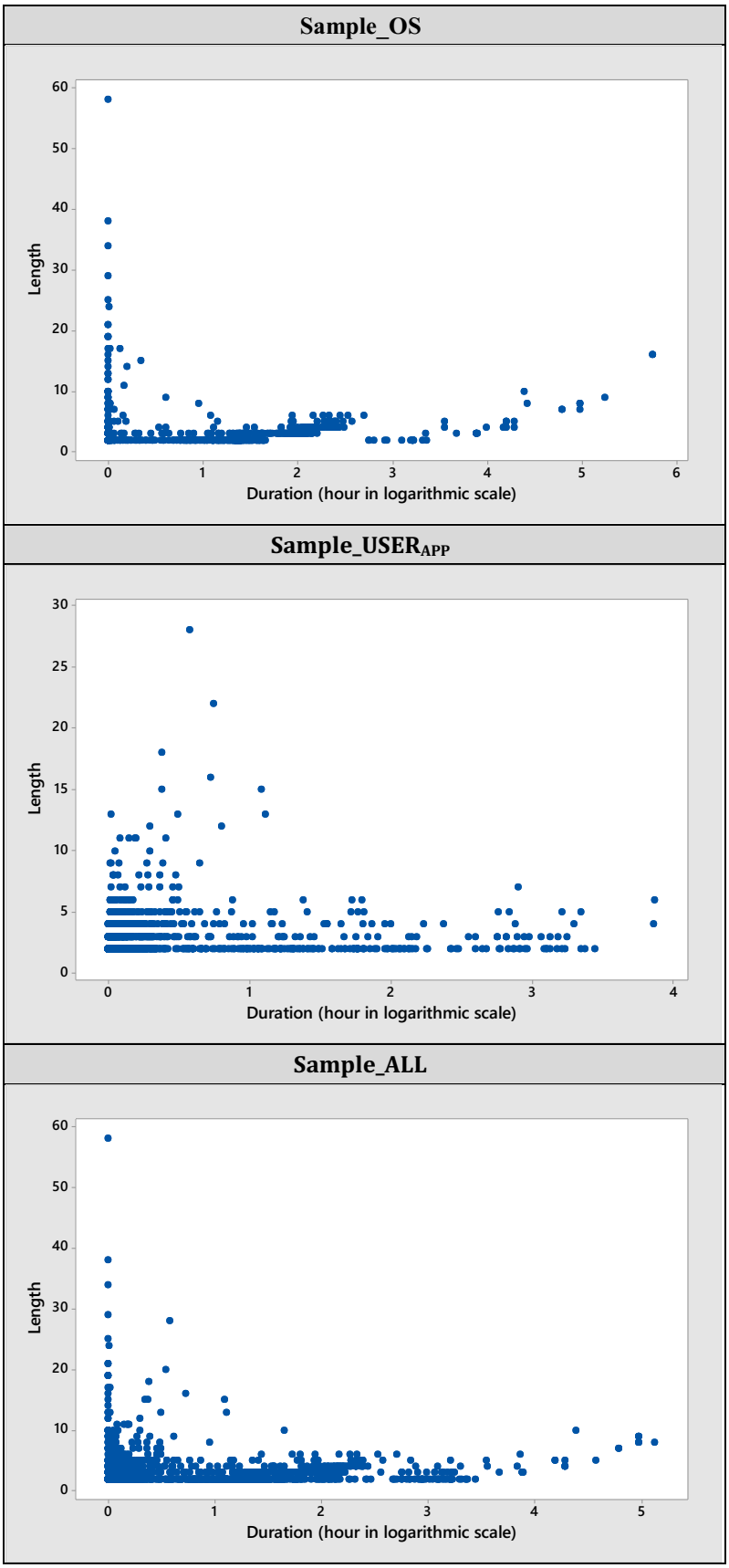


Figure 5.6. Sequence length vs time duration.

The sequences within the first interval (i.e., 0 to 0.22 hours) of the Sample_OS are highly diverse, encompassing 124 different sequences with a total of 224 occurrences. The most recurrent sequence observed in this interval is $WinU^{800b0100} \rightarrow WinU^{800b0100}$ with 8.22% of the occurrences. This most frequent sequence indicates a repetition of failures due to corruption of files needed for the WUS to update the Win7 core components. Furthermore, it is also observed a high variety of sequence lengths in this interval, in which the most common (54.46%) combinations are no longer than two OS failures. On the other hand, in the second interval (i.e., 2.32 to 3.06 hours), I only observed six different sequences (187 occurrences in total), in which the most prevalent (89.30%) have the configuration $IEU^{80070643} \rightarrow IEU^{80070643}$, which indicates successive failures during the installation of Internet Explorer updates caused by malfunctions in the .Net Framework software. Sequences with two OS failures prevailed (94.12%) in the second interval.

For the time interval between 0 and 0.11 hours, I found 228 sequences with 1,092 occurrences in the Sample_$USER_{APP}$. The most frequent sequence within the above-mentioned interval is $iexplore^{0xc0000005} \rightarrow iexplore^{0xc0000005}$ with 15.75% of occurrences. Moreover, most sequences in the interval are composed of two failures (74.45%).

Looking for patterns combining sequence lengths and sequence durations, I analyzed the scatterplots shown in Figure 5.6. Note that, in Sample_OS, the majority of sequences with more than three failures present shorter durations when compared to the majority of smaller sequences. Thus, at least for the longer OS sequences, this finding corroborates my previous assumption of a common cause for the multiple failures in a sequence (see Section 5.3.2). On the other hand, longer $USER_{APP}$ are concentrated within the interval 0.4 (24 minutes) to 2 hours.

5.4 Failure Prediction

In this thesis, the approach proposed for software failure prediction uses patterns of failure sequences as input to the predictive models. To do that, the two required input raw data necessary are the 'failure type' and the 'failure event timestamp'. The failure type is any information that allows identifying a software failure, such as a numerical code (e.g., *EventIdentifier* in Win7, see Table 3.2), the canonical name of the failed software/component (e.g., *explorer.exe*), the category of the failed software (e.g., OS or $USER_{APP}$), and so forth. These two required fields are present in most system logs, making this approach as general as possible. Different from the sequence analysis discussed in sections 4.3 and 5.3, for the failure prediction I avoid using very specific data in the models, like the 'failure causes', which are not easily found in system logs.

5.4.1 Failure Prediction Based on Sequences of Two Failure Categories

To conduct a higher-level analysis using the whole dataset (Sample_ALL), I set the failure categories considered in this study (i.e., OS_{KNL}, OS_{SVC}, OS_{APP}, and $USER_{APP}$) as the four possible values of the dependent variable (see Section 4.4) and compute their probability of occurring given the occurrence of other failures immediately before them (independent variable). The independent variable also

assumes one of the failure category values. Table 5.10 shows all possible sequential associations found between the values of the dependent (Last Failure) and independent (Previous Failure) variables, and the upper bounds of the confidence intervals for the median value of their Δts. Note that the highest upper bounds are related to OS_{KNL} failures. Moreover, the association between OS_{KNL} failures with failures from other categories amounted to only 1.91% of all associations analyzed. This low percentage may be due to the effect of system reboots that regularly occur right after OS_{KNL} failures, lowering the likelihood of nearby failures. Besides, kernel subsystems run at the kernel space, thus in a protected operating mode, which makes it less likely to be affected by failures from other categories that run at the User space.

Next, as mentioned in Section 4.4, I select all occurrences of the failure sequences whose Δts are less or equal to the upper bound value of their respective median's confidence interval; otherwise, I remove them from the work sample. Subsequently, to decrease the bias of selecting between the training and test data sets, I randomly split the work sample into 70% and 30% respectively.

Since the work sample contains all possible combinations between the dependent and independent variables (i.e., every failure category occurred along with the others), it is not necessary to apply regularization to the Multinomial Logistic Regression (see Section 2.1.4). Therefore, equations 11, 12, and 13 show the three resulting logits that model the four failure categories on both dependent and independent variables. Note that the OS_{APP} category was chosen as the reference value for the dependent variable and it was omitted on the dummy variables created for the categorical independent variable.

Table 5.10. Sequential associations based on failure categories.

Sequential Association		95% CI for the Median of Δts		
Independent Variable (Previous Failure)	**Dependent Variable (Last Failure)**	**Lower (hour)**	**Median (hour)**	**Upper (hour)**
OS_{KNL}	OS_{KNL}	0.276	1.143	1.521
OS_{KNL}	OS_{SVC}	0.234	1.755	10.483
OS_{KNL}	OS_{APP}	16.445	22.855	31.252
OS_{KNL}	$USER_{APP}$	1.390	5.520	7.614
OS_{SVC}	OS_{KNL}	1.467	7.207	17.467
OS_{SVC}	OS_{SVC}	2.432	2.636	2.865
OS_{SVC}	OS_{APP}	0.129	1.508	5.978
OS_{SVC}	$USER_{APP}$	5.368	6.234	7.814
OS_{APP}	OS_{KNL}	1.442	6.943	24.006
OS_{APP}	OS_{SVC}	0.823	2.803	4.419
OS_{APP}	OS_{APP}	0.001	0.003	0.011
OS_{APP}	$USER_{APP}$	2.255	4.638	6.823
$USER_{APP}$	OS_{KNL}	9.999	13.934	21.594
$USER_{APP}$	OS_{SVC}	1.827	2.448	2.664
$USER_{APP}$	OS_{APP}	1.042	2.632	3.328
$USER_{APP}$	$USER_{APP}$	0.115	0.130	0.143

$$g_{KNL} = ln\left(\frac{Pr(Y = OS_{KNL}|x_{KNL},\, x_{SVC},\, x_{USERAPP})}{Pr(Y = OS_{APP}|x_{KNL},\, x_{SVC},\, x_{USERAPP})}\right) = -3.00 + 5.24x_{KNL} + 2.92x_{SVC} + 3.15x_{USERAPP} \quad (11)$$

$$g_{SVC} = ln\left(\frac{Pr(Y = OS_{SVC}|x_{KNL},\, x_{SVC},\, x_{USERAPP})}{Pr(Y = OS_{APP}|x_{KNL},\, x_{SVC},\, x_{USERAPP})}\right) = -2.66 + 2.66x_{KNL} + 7.16x_{SVC} + 3.24x_{USERAPP} \quad (12)$$

$$g_{USERAPP} = ln\left(\frac{Pr(Y = USER_{APP}|x_{KNL},\, x_{SVC},\, x_{USERAPP})}{Pr(Y = OS_{APP}|x_{KNL},\, x_{SVC},\, x_{USERAPP})}\right) = -0.69 + 2.79x_{KNL} + 3.26x_{SVC} + 5.36x_{USERAPP} \quad (13)$$

The McFadden's pseudo R^2 value calculated for the model is 0.55, which indicates very good adherence of the model to the training data set [53]. Moreover, evaluating the model's predictions against the test data set, it obtains a prediction accuracy of 93.88%. Both Decision Tree and Random Forest models have the same accuracy result.

In order to have a closer view of how the three models reach such accuracy, I calculated the probabilities of all dependent variable values based on the values of the independent variable (previous failure event). The Multinomial Logistic Regression and Decision Tree models presented the same values of probabilities. Figure 5.7 shows the probabilities obtained by both models. The letters K, S, A, and U on both axis represent the failures categories OS_{KNL}, OS_{SVC}, OS_{APP}, and $USER_{APP}$, respectively. The blue color indicates a low probability (around 0%) of occurring a certain failure given the occurrence of a previous failure while the red color indicates a high probability (around 100%). Moreover, the grey color refers to a medium probability of occurrence (around 50%). The Random Forest model has a slight difference in the probabilities compared to the Multinomial Logistic Regression and Decision Tree models (see Figure A.1 in the Appendix).

Note that the probability is higher when both last (dependent variable) and previous (independent variable) failures have the same category, especially for $USER_{APP}$ and OS_{SVC} failures. This finding indicates that it is more likely that a failure occurs along with other failures of the same failure category, regardless of the category. I conjecture that this failure pattern explains the high McFadden's pseudo R^2 value observed in the Multinomial Logistic Regression, given that the dependent variables were almost perfectly predicted in cases where the independent and dependent variables had the same values.

However, as shown in the last row of Figure 5.7, there is also a significantly higher probability of occurring $USER_{APP}$ failures after all other categories, especially after OS_{KNL} and OS_{APP} failures. This may have happened due to the high prevalence of $USER_{APP}$ failures in the dataset (83.39%) compared to the other categories (i.e., OS_{KNL}(3.66%), OS_{SVC} (10.07%), OS_{APP} (2.88%)).

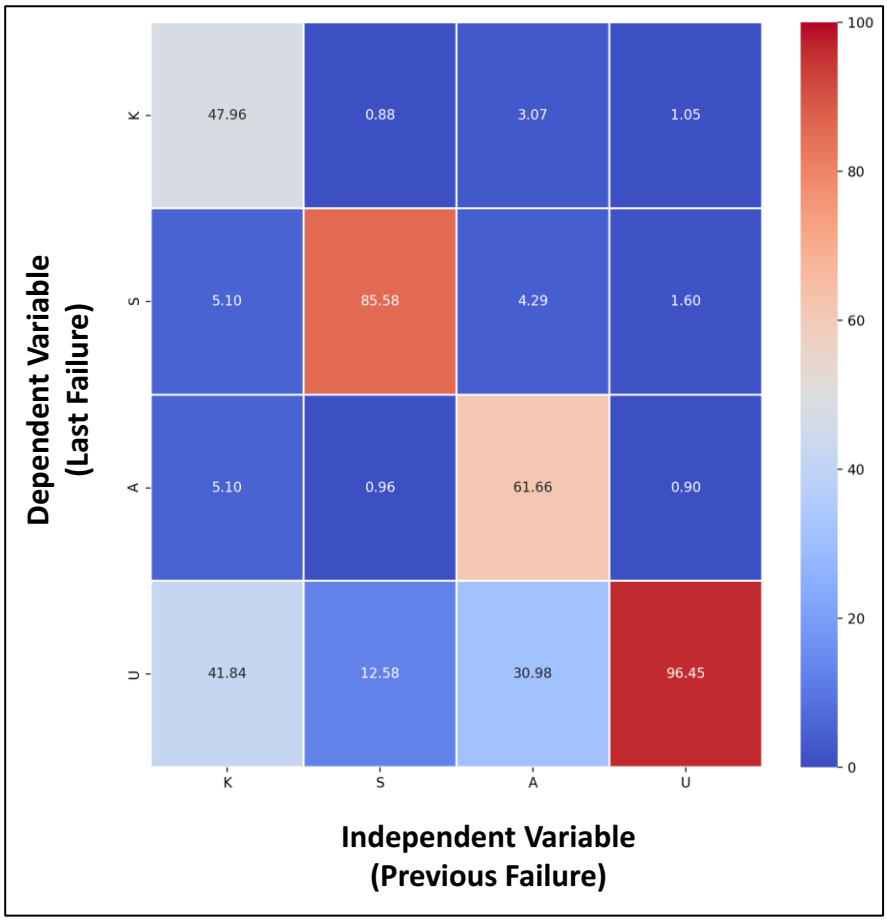


Figure 5.7. Probabilities based on sequences of two failure categories (Multinomial Logistic Regression and Decision Tree).

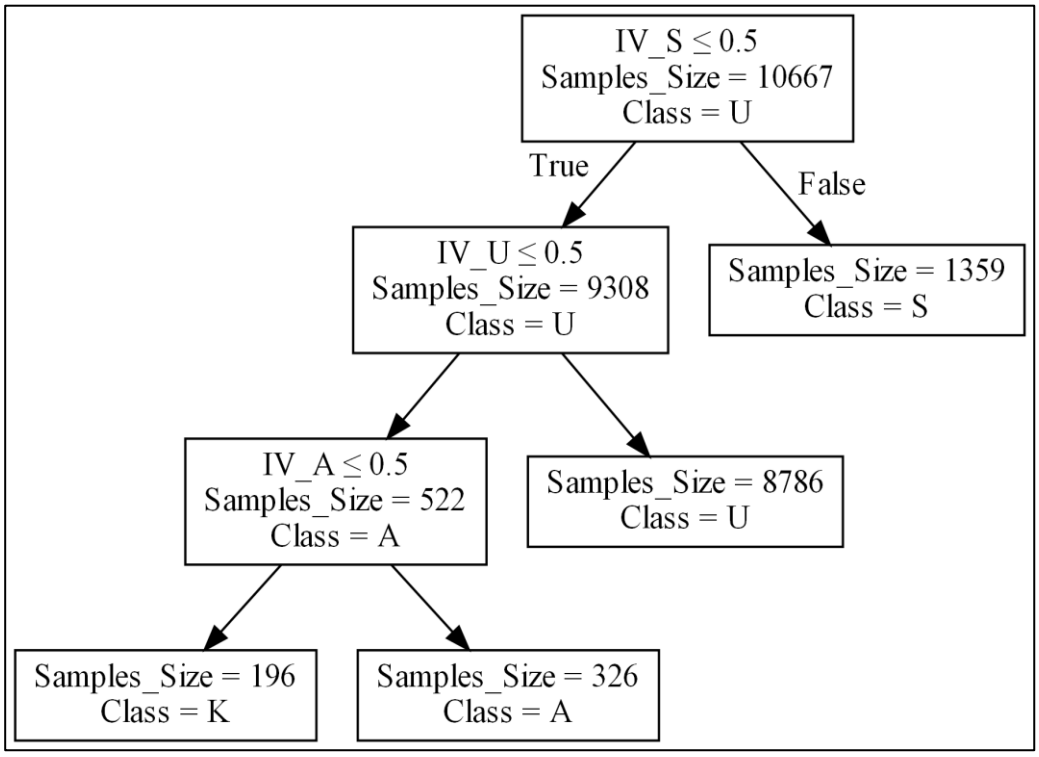


Figure 5.8. Decision Tree based on sequences of two failure categories.

Figure 5.8 presents the resulting Decision Tree. Note that the internal nodes, which represent the independent variables (Previous Failure), have three attributes. The only attribute Leaf nodes do not have is the first, which indicates the decision based on the independent variable. The three attributes of the internal nodes are:

1. **IV_* ≤ 0.5**: indicates the decision based on the independent variable (IV), which, in this case, is whether the previous failure is of a certain category in a sequence of two failures. For example, the decision in the Root node, i.e., IV_S ≤ 0.5, indicates that if the previous failure is not an OS_{SVC} failure (i.e., IV_S = 0) go to the left edge (IV_S ≤ 0.5 is True), and if the previous failure is an OS_{SVC} failure (i.e., IV_S = 1) go to the right edge (IV_S ≤ 0.5 is False). Note that the independent variables are either 0 or 1 since they are dummy variables.

2. **Sample_Size**: indicates the number of training data in the node. Therefore, the Root node contains the whole training sample. The other nodes segment the training data based on the decisions. For example, the internal node to the left of the Root node shows that 9,308 of the 10,667 sequences in the training data do not have an OS_{SVC} failure as previous failure.

3. **Class**: indicates the dependent variable value with the highest occurrence for the training data in a node. For example, the Root node shows that most sequences in the training sample have a $USER_{APP}$ failure as the last failure.

Note that the last attribute of the Leaf nodes (Class) indicates the predicted outcome of the dependent variable. Therefore, the Decision Tree shown in Figure 5.8 can be interpreted as a series of "ifs and elses":

- If the previous failure is OS_{SVC}, the next failure is OS_{SVC}
- Else if the previous failure is $USER_{APP}$, the next failure is $USER_{APP}$
- Else if the previous failure is OS_{APP}, the next failure is OS_{APP}
- Else the next failure is OS_{KNL}

Observe that the Decision Tree also depicts the pattern where sequences having both previous and last failure of the same category are more probable of occurring in the sample, forming a long ramification to the left side of the tree. Moreover, it is not necessary to include an internal node (independent variable) to represent OS_{KNL} failures, because if the previous failure of a sequence is neither of the other three failure categories (i.e., OS_{SVC} or $USER_{APP}$ or OS_{APP}), it is OS_{KNL}.

Finally, regarding the number of trees in the Random Forest model, of the seven values tested (10, 29, 49, 69, 88, 108, and 128), the best amount of trees in the model was 10 (default value).

5.4.2 Failure Prediction Based on Sequences of Two Failure Types

Next, I followed the same rationale used in Section 5.4.1, in which I computed the probability of failures to occur (dependent variable) given the occurrence of other failures immediately before (independent variable). However, in this case, I set sequences of two failure Types (instead of failure Categories) as input to the models. Moreover, I considered sequences from each of the three samples adopted in this study — the first sample containing only OS failures (Sample_OS), the second containing only User Application failures (Sample_$USER_{APP}$), and the third represents the whole dataset i.e., both OS and User Application failures (Sample_ALL).

Given the sparsity problem (see Section 2.1.4) that emerges when dealing with the great number of failure types, it is necessary to use the Multinomial Logistic Regression with Ridge regularization. The following equations present a general representation of the Multinomial Logistic Regression with Ridge regularization models needed to represent the failure types on both dependent and independent variables.

$$Pr(Y = f_1 | X_{before} = \mathbf{x_b}) = \frac{e^{\beta_{0f_1} + \boldsymbol{\beta}_{f_1}^T \mathbf{x_b}}}{\sum_{l=f_1}^{f_k} e^{\beta_{0l} + \boldsymbol{\beta}_l^T \mathbf{x_b}}}$$

...

$$Pr(Y = f_k | X_{before} = \mathbf{x_b}) = \frac{e^{\beta_{0f_k} + \boldsymbol{\beta}_{f_k}^T \mathbf{x_b}}}{\sum_{l=f_1}^{f_k} e^{\beta_{0l} + \boldsymbol{\beta}_l^T \mathbf{x_b}}}$$

where the dependent variable $Y = \{ f_1, \ldots, f_k \}$ has k levels (i.e., k possible failure types). Note that an equation is created for each level of the dependent variable and the probability of each level is divided by the probability of all other levels. X_{before} represents the failure event immediately preceding the last failure (dependent variable) of the sequences and $\mathbf{x_b}$ is a vector of dummy independent variables of all possible failure types that immediately precede the last failure. Due to the regularization, I did not calculate the McFadden's pseudo R^2, which measures how well the models fitted the training data since the regularization introduces a penalization that makes the model less sensitive to the training data.

By evaluating the three (i.e., Multinomial Logistic Regression with Ridge regularization, Decision Tree, and Random Forest) models' predictions against the test data set, I obtained the same prediction accuracy of 86.99% for the Sample_OS, 90.89% for the Sample_$USER_{APP}$, and 88.28% for the Sample_ALL. To find an explanation of why three different models are having the same accuracy, figures 5.9, 5.10, and 5.11 show the probabilities calculated by each model for sequences in the Sample_ALL. As mentioned in Section 2.1.4, the Ridge regularization introduces a penalization to the model's parameters forcing the less relevant ones to be close, but never reach zero. Therefore, all possible combinations between the dependent and independent variables have a probability. On the other hand, both Decision Tree and Random Forest models only calculate the probability of combinations present in the training sample. Consequently, the blank (whitespaces) probabilities in Figures 5.10 and 5.11 indicate combinations that were not observed in the training data and the models could not calculate them.

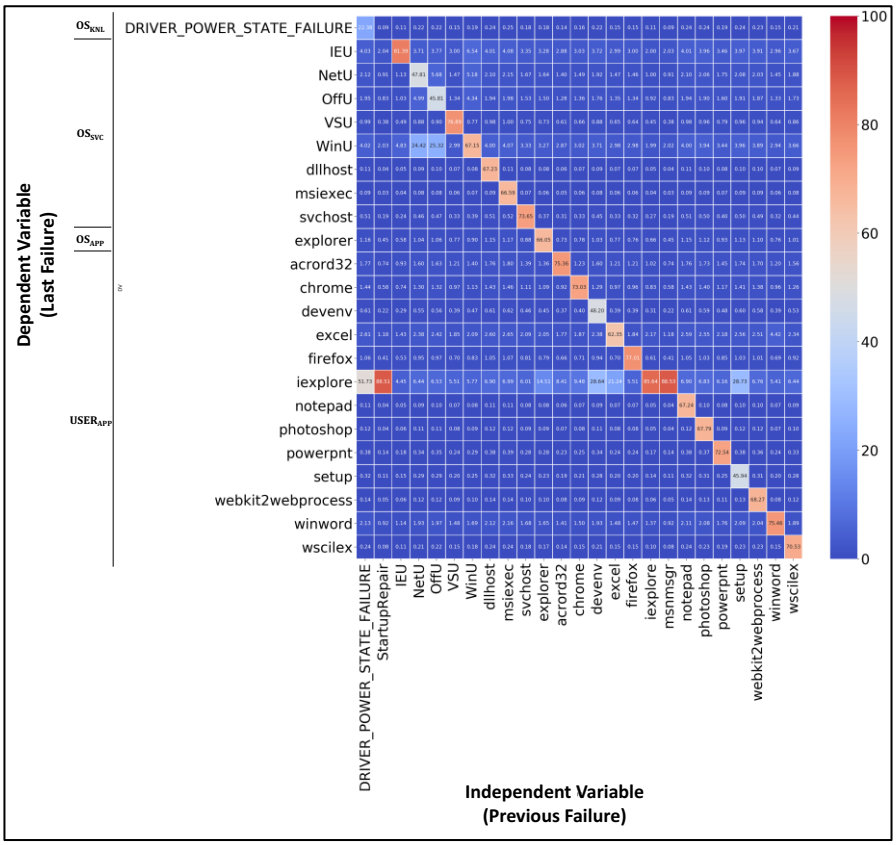


Figure 5.9. Probabilities based on sequences of two failure types (Multinomial Logistic Regression with Ridge - Sample_ALL).

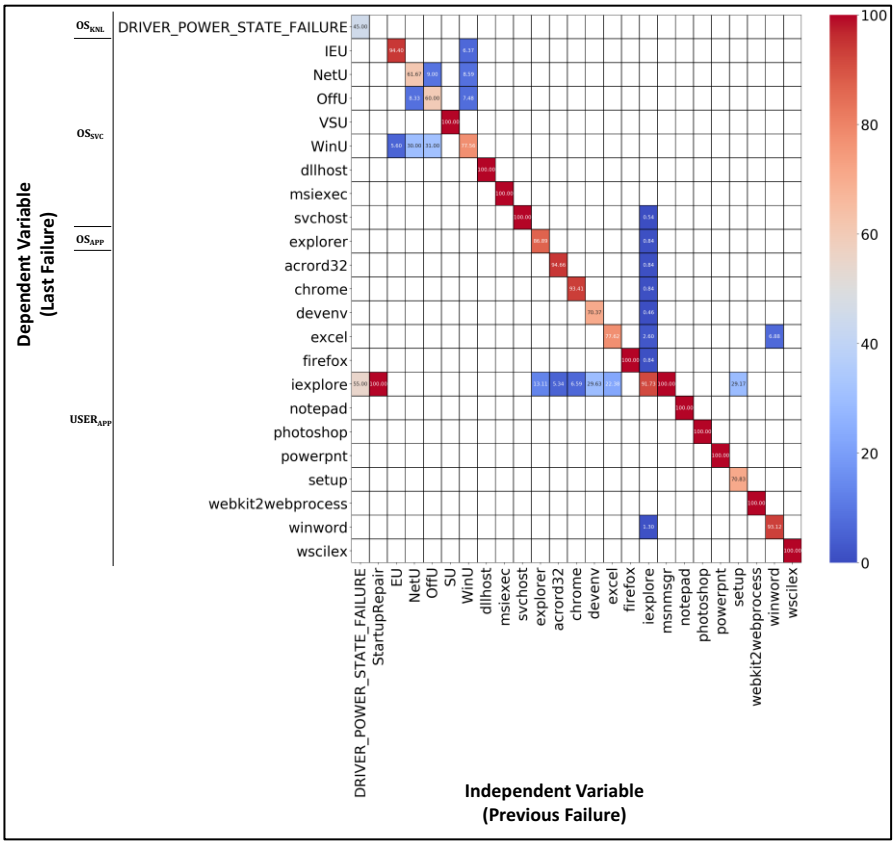


Figure 5.10. Probabilities based on sequences of two failure types (Decision Tree - Sample_ALL).

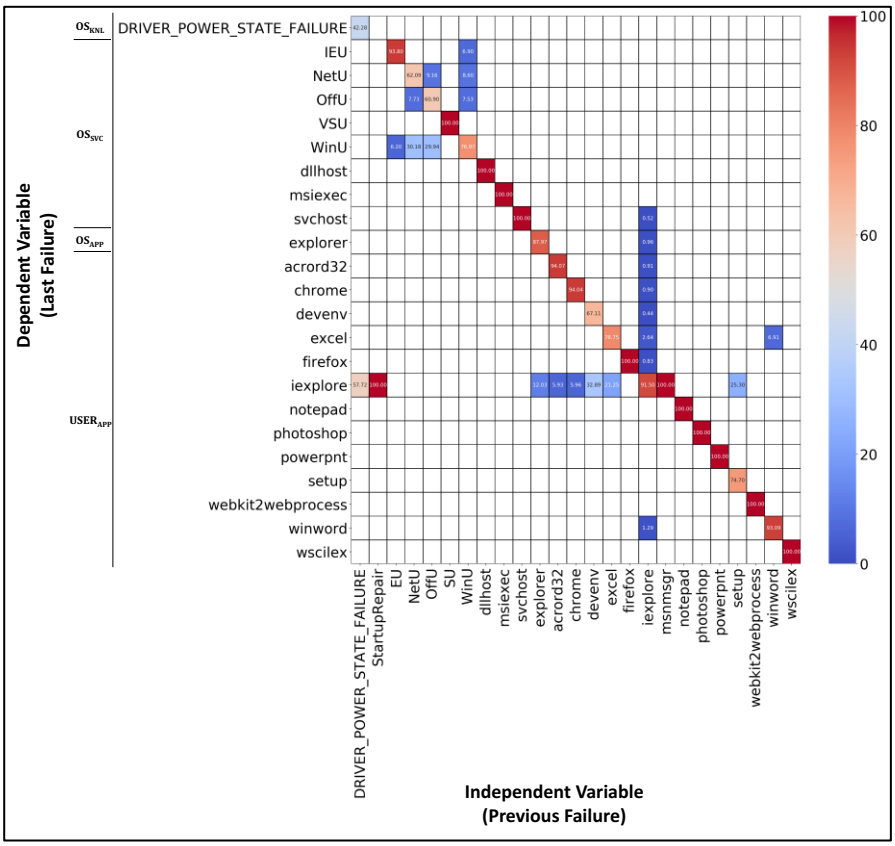


Figure 5.11. Probabilities based on sequences of two failure types (Random Forest - Sample_ALL).

Note that the same pattern observed in Figure 5.7 in which the main diagonal has the highest probabilities is also observed in figures 5.9, 5.10, and 5.11. Therefore, as observed in Section 5.4.1 for failure Categories, there is also a high pattern of multiple same-type failures on the sequences of two failure types analyzed. Thus, a possible explanation of why the models are having the same accuracy is the simplicity of their inputs (sequences with only two failures), together with the strong pattern of multiple same-type failures.

Besides the pattern of multiple same-type failures observed, it is important to emphasize that sequences of *iexplore* ($USER_{APP}$) occurring after OS_{KNL} failures (*DRIVER_POWER_STATE_FAILURE* and *StartupRepair*) are also observed. However, they are among the least frequent sequences found with 0.28% and 0.24% of the occurrences. The same is observed for *devenv→iexplore* (0.24%) and *setup→iexplore* (0.19%). Therefore, these sequences may have occurred due to the predominance of *iexplore* failures in the dataset (see Table 3.5).

Figures A.2 to A.7, in the Appendix, present the probabilities obtained for Sample_OS and Sample_$USER_{APP}$, respectively, using the three proposed models. The same pattern of the highest probabilities occurring in the main diagonal is also observed in both OS and $USER_{APP}$ samples. Moreover, Figures A.8 to A.10, in the Appendix, show the Decision Trees for each of the three samples considered in this study. All trees presented the same long ramification to the left side of the tree as observed in Figure 5.8, which also indicates patterns of multiple same-type

failures. Finally, similar to Section 5.4.1, the best amount of trees in the Random Forest model was 10 (default value) for all of the three samples analyzed.

5.4.3 Failure Prediction Based on Sequences of Two Failure Types and the Δt between them

One of the most important properties that provide a better assessment of how the failure sequences occur is their Δt values. Hence, I extend the analyses presented in Section 5.4.2 by including the Δt of the failure sequences as an additional independent variable (IV) to the multinomial models. First, the sequences' Δts are categorized by computing the minimum, median, and maximum of their values.

Table 5.11 shows the Δt statistics for each sample. Due to the continuous nature of the Δt values, I group them into two classes, "Minimum ≤ Δt ≤ Median" and "Median < Δt ≤ Maximum", in order to use them as input in the prediction models. It is noteworthy that the time precision recorded by the RAC instrumentation is one second. So, if two failure events occur within a Δt < 1 s they are registered with the same timestamp, which explains the occurrences of minimum values of Δt equal to 0.

Analyzing the whole dataset, most sequences with the highest median Δts are composed of *iexplore* failures with other failure types. In fact, of the nine sequences with the longest Δt median, only two (i.e., *DRIVER_POWER_STATE_FAILURE→ DRIVER_POWER_STATE_FAILURE* and *VSU→VSU*) do not include *iexplore* failures, which strengthen the hypothesis that these sequences may have occurred due to the predominance of *iexplore* failures in the dataset. Once again, I use the Multinomial Logistic Regression with Ridge regularization because of the great number of values in the dependent and independent variables, causing the sparsity problem. The following equations depict the regression with the addition of the Δts as a second independent variable.

$$Pr(Y = f_1 | X_{before} = \mathbf{x_b}, X_{\Delta t} = \mathbf{x_{\Delta t}}) = \frac{e^{\beta_{0f_1} + \boldsymbol{\beta}_{f_1}^T \mathbf{x_b} + \boldsymbol{\beta}_{f_1}^T \mathbf{x_{\Delta t}}}}{\sum_{l=f_1}^{f_k} e^{\beta_{0l} + \boldsymbol{\beta}_l^T \mathbf{x_b} + \boldsymbol{\beta}_l^T \mathbf{x_{\Delta t}}}}$$

...

$$Pr(Y = f_k | X_{before} = \mathbf{x_b}, X_{\Delta t} = \mathbf{x_{\Delta t}}) = \frac{e^{\beta_{0f_k} + \boldsymbol{\beta}_{f_k}^T \mathbf{x_b} + \boldsymbol{\beta}_{f_k}^T \mathbf{x_{\Delta t}}}}{\sum_{l=f_1}^{f_k} e^{\beta_{0l} + \boldsymbol{\beta}_l^T \mathbf{x_b} + \boldsymbol{\beta}_l^T \mathbf{x_{\Delta t}}}}$$

Table 5.11. Summary of the sequences' durations (in hours).

	OS	$USER_{APP}$	All Failures
Minimum	0.000	0.000	0.000
Median	0.177	0.019	0.019
Maximum	28.668	30.378	30.378

In this case, the accuracy of the Multinomial Logistic Regression with Ridge regularization models are slightly different from the Decision Tree and Random Forest models applied to all samples. Table 5.12 presents the accuracies from each model per sample. The numbers in parenthesis indicate how much the accuracy increased (+) or decreased (-) in relation to the analysis performed in Section 5.4.2.

Note that the accuracies of all models do not change much (e.g., the maximum change is 0.46% in the accuracy) in relation to the ones observed in Section 5.4.2. However, I discovered new other failure sequence patterns by analyzing the calculated probabilities of all models. Figures 5.12, 5.13, and 5.14 show the probabilities calculated by the Logistic Regression with Ridge regularization, Decision Tree, and Random Forest models, respectively, for sequences in the Sample_OS.

Table 5.12. Summary of the models' accuracy.

	OS	$USER_{APP}$	All Failures
Model	**Accuracy**	**Accuracy**	**Accuracy**
MLR with Ridge	(0.00) 86.99%	(-0.46) 90.43%	(-0.43) 87.85%
Decision Tree	(+0.35) 87.34%	(-0.46) 90.43%	(+0.44) 88.72%
Random Forest	(+0.35) 87.34%	(-0.46) 90.43%	(+0.44) 88.72%

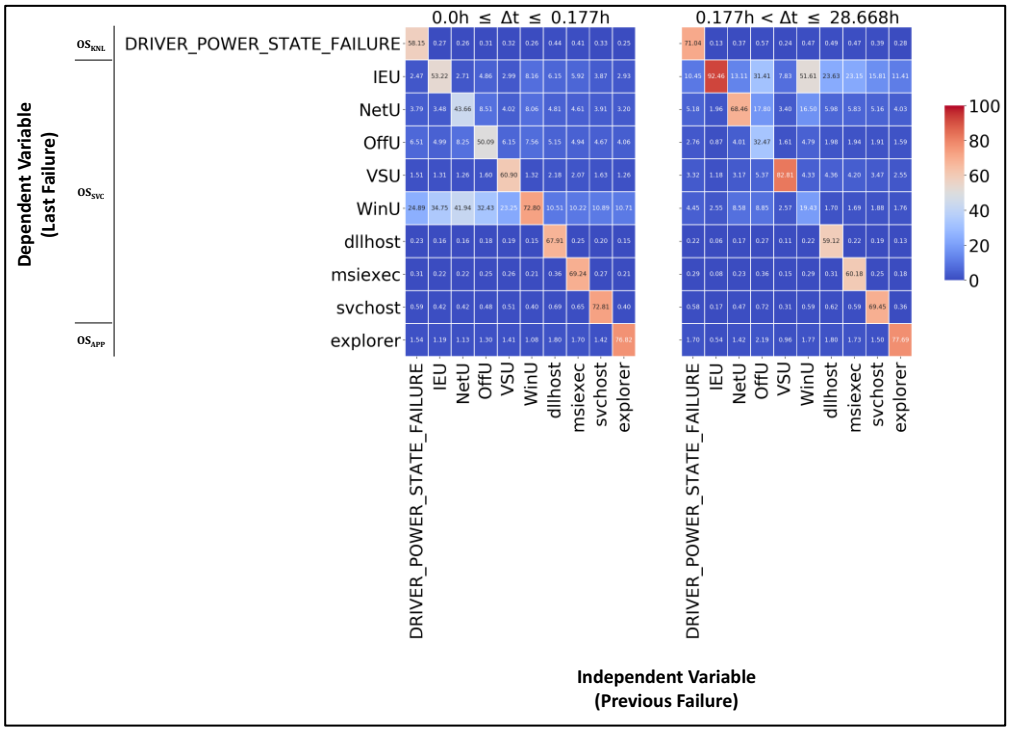


Figure 5.12. Probabilities based on sequences of two failure types and the Δt between them (Multinomial Logistic Regression with Ridge - Sample_OS).

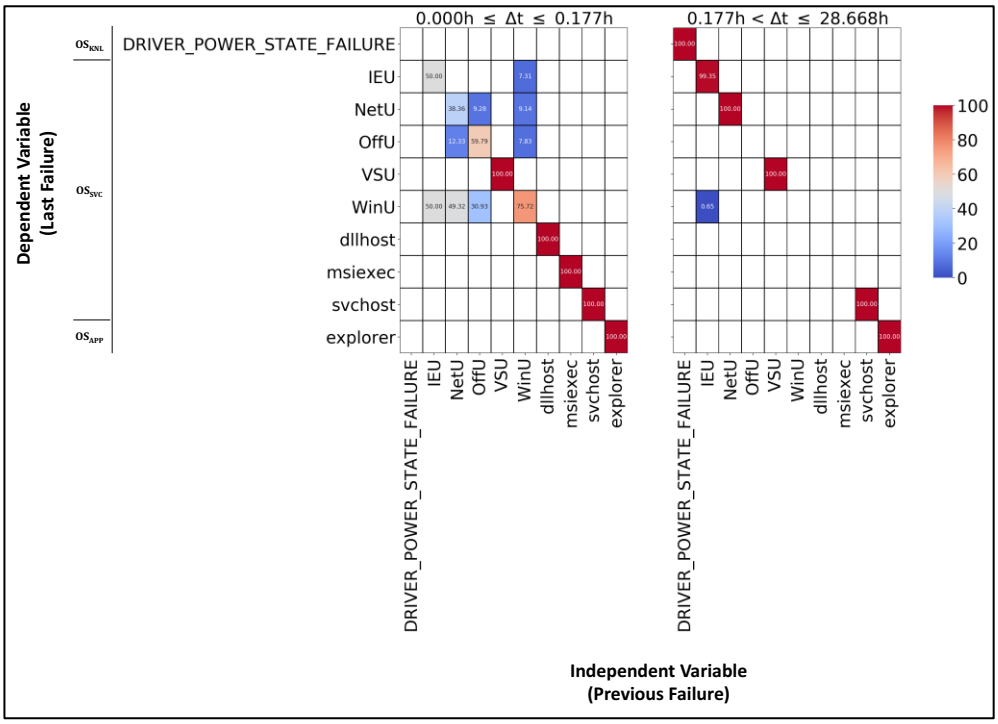


Figure 5.13. Probabilities based on sequences of two failure types and the Δt between them (Decision Tree - Sample_OS).

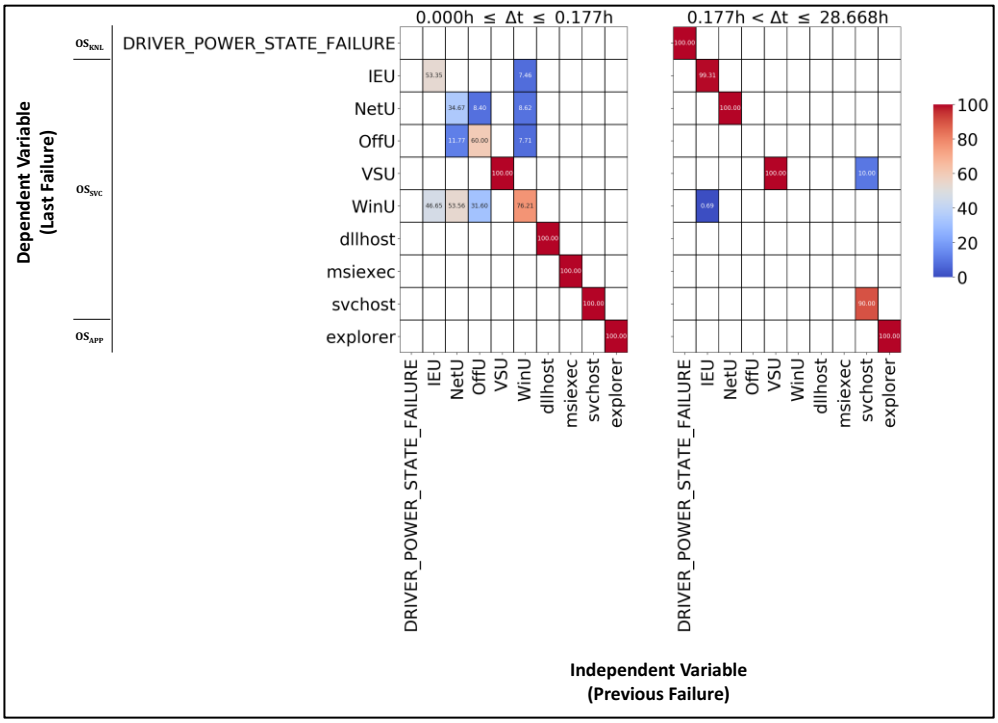


Figure 5.14. Probabilities based on sequences of two failure types and the Δt between them (Random Forest - Sample_OS).

Note that the above figures present two heatmaps — the first displays the probabilities of sequences within the interval "Minimum ≤ Δt ≤ Median" and the second displays the probabilities of sequences within the interval "Median < Δt ≤ Maximum". New failure sequence patterns were found based on the calculated probabilities. For example, note that sequences with *Windows Update* failures (*WinU*) are more probable to occur after most other update failures in smaller Δts,

which is depicted by figures 5.12, 5.13, and 5.14 as a lighter strip of squares for the *WinU* (y-axis) where Δts are between 0.0 h and 0.177 h. On the opposite side, *Internet Explorer Update* failures (*IEU*) have a high probability of occurring in longer Δts. Note that some probabilities in certain intervals could not be calculated by both Decision Tree and Random Forest models, because both models can only compute factual probabilities of sequences that occur in the training sample. For example, the probability of occurring a *DRIVER_POWER_STATE_FAILURE* failure after other *DRIVER_POWER_STATE_FAILURE* is high for both Δt intervals in the models of Multinomial Logistic Regression with Ridge (see Figure 5.12). However, for the first Δt interval (i.e., 0.000 h ≤ Δt ≤ 0.177 h), in figures 5.13 and 5.14, no probability is calculated since the training sample does not have any occurrence of the sequence *DRIVER_POWER_STATE_FAILURE→DRIVER_POWER_STATE_FAILURE* in the first interval. Figure 5.15 presents the Decision Tree created from the Sample_OS, which shows that the resulting Decision Tree does not have any *DRIVER_POWER_STATE_FAILURE* leaf node in the first interval, explaining the lack of probabilities in the above-mentioned figures. Figures A.11 and A.12, in the Appendix, show the Decision Trees created from Sample_USER$_{APP}$ and Sample_ALL, respectively. All Decision Trees continue to present patterns of multiple same-type failures having a long ramification to the left side.

Figures A.13 to A.18 (in Appendix) present the calculated probabilities for the Sample_USER$_{APP}$ and Sample_ALL, respectively. Regarding User Application failures, I observe a higher probability of occurring *iexplore* failures with other failure types in longer Δts, as observed at the beginning of this section. Finally, once more, the best amount of trees in the Random Forest models is 10 (default) for Sample_OS and Sample_USER$_{APP}$. However, this number changes to 29 for Sample_ALL.

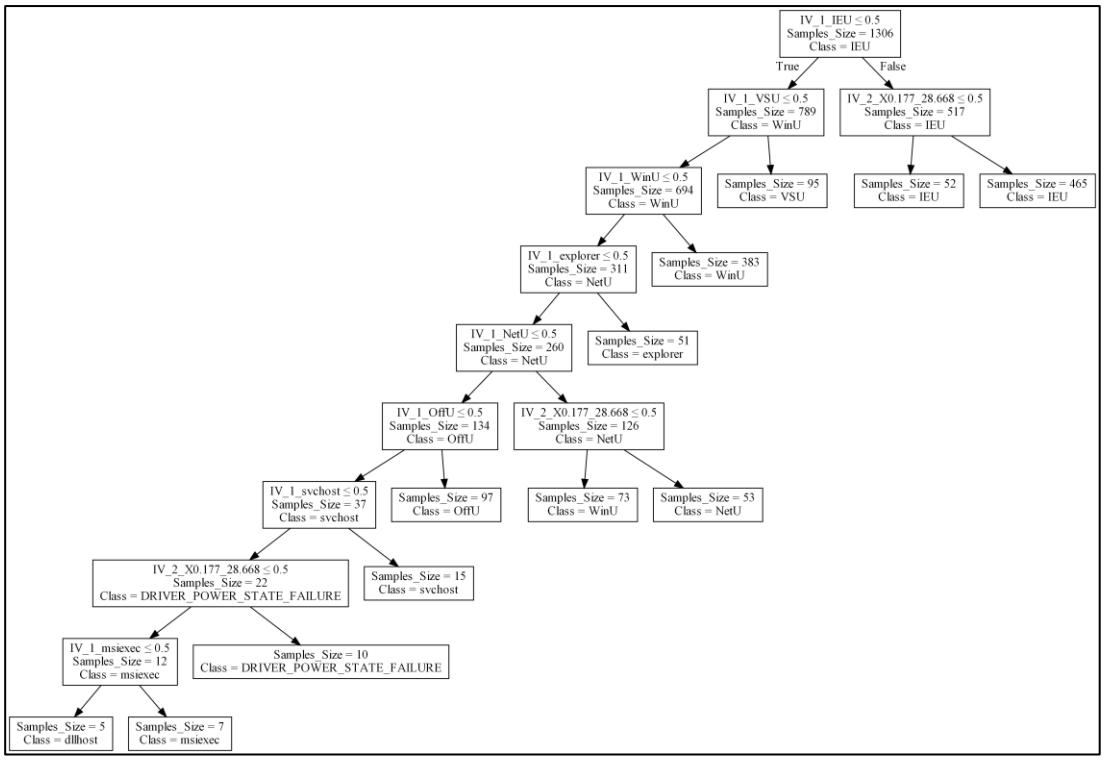


Figure 5.15. Decision tree based on sequences of two failure types and the Δt between them (Sample_OS).

5.4.4 Failure Prediction Based on Sequences of Three Failure Types

Since one of the most common lengths observed in the failure sequences analyzed in this study is three (see Section 5.3.3), I also calculate the probability of a failure to occur (dependent variable) given the occurrence of other two preceding failures (independent variables). The following equations represent the Multinomial Logistic Regression with Ridge regularization models needed to represent this type of sequence:

$$Pr(Y = f_1 | X_{before\_1} = \mathbf{x_{b1}}, X_{before\_2} = \mathbf{x_{b2}}) = \frac{e^{\beta_{0f_1} + \boldsymbol{\beta}_{f_1}^T \mathbf{x_{b1}} + \boldsymbol{\beta}_{f_1}^T \mathbf{x_{b2}}}}{\sum_{l=f_1}^{f_k} e^{\beta_{0l} + \boldsymbol{\beta}_l^T \mathbf{x_{b1}} + \boldsymbol{\beta}_l^T \mathbf{x_{b2}}}}$$

...

$$Pr(Y = f_k | X_{before\_1} = \mathbf{x_{b1}}, X_{before\_2} = \mathbf{x_{b2}}) = \frac{e^{\beta_{0f_k} + \boldsymbol{\beta}_{f_k}^T \mathbf{x_{b1}} + \boldsymbol{\beta}_{f_k}^T \mathbf{x_{b2}}}}{\sum_{l=f_1}^{f_k} e^{\beta_{0l} + \boldsymbol{\beta}_l^T \mathbf{x_{b1}} + \boldsymbol{\beta}_l^T \mathbf{x_{b2}}}}$$

where X_{before_1} represents the failure event immediately preceding the last failure (dependent variable) of the sequences and X_{before_2} the one that occurred previously to the X_{before_1}. Table 5.13 presents the accuracies from each model per sample. Note that all models present similar accuracies, where the greatest difference among them is 2.34% between the highest accuracy (88.28%) obtained by the Multinomial Logistic Regression with Ridge regularization and the lowest (85.94) obtained by the Random Forest in the Sample_OS.

Figures 5.16, 5.17, and 5.18, present the probabilities calculated for the Logistic Regression with Ridge regularization, Decision Tree, and Random Forest models, respectively, based on Sample_ALL. The x-axis represents the X_{before_1} failures and each graph indicates one possible value for X_{before_2}. Given the high number of X_{before_2} values, I filter the data to present only the most frequently observed in the sample. Note that that the number of failure types decreased given that several types do not occur in sequences composed of three failures. Moreover, when the first failure (X_{before_2}) of the sequences occurs, the probability of the last failure (dependent variable) having the same type as the first increases. There are also cases where the last failures have a high probability of being the same type as its closest failure (X_{before_1}). Both cases occur due to the pattern of multiple same-type failures observed throughout the prediction analysis. However, there is also a significant probability of occurring *iexplore* failures (see Figure 5.16), which, as stated in the previous sections, may occur due to the high frequency of this failure in the dataset. Figures A.19 to A.27 (in the Appendix) present all calculated probabilities from each model per sample.

Table 5.13. Models' accuracy (sequences of three failure types).

	OS	$USER_{APP}$	All Failures
Model	**Accuracy**	**Accuracy**	**Accuracy**
MLR with Ridge	88.28%	92.50%	92.31%
Decision Tree	86.72%	91.58%	92.01%
Random Forest	85.94%	91.58%	92.33%

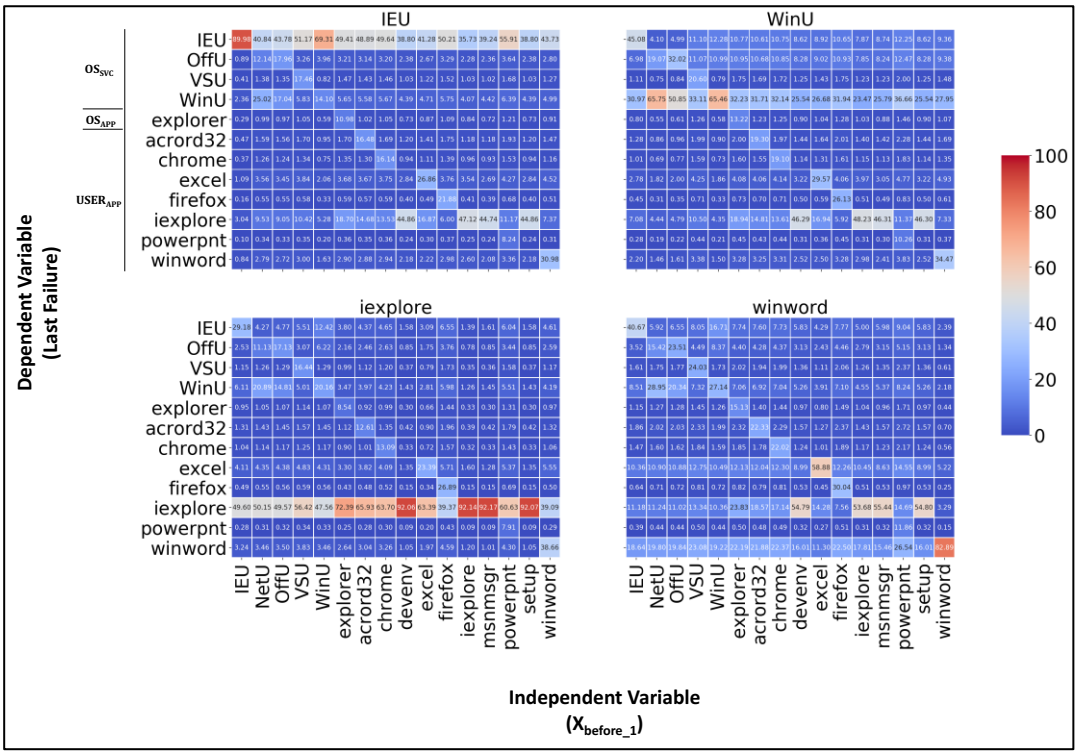


Figure 5.16. Probabilities based on sequences of three failure types (Multinomial Logistic Regression with Ridge – Filtered Sample_ALL).

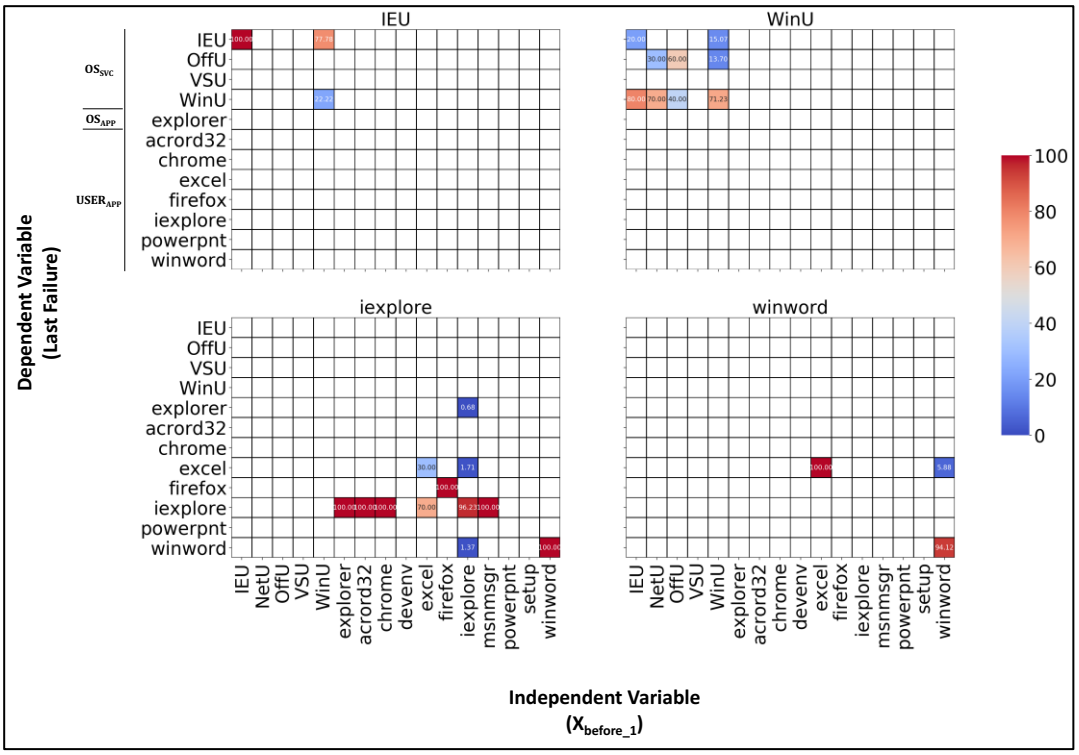


Figure 5.17. Probabilities based on sequences of three failure types (Decision Tree – Filtered Sample_ALL).

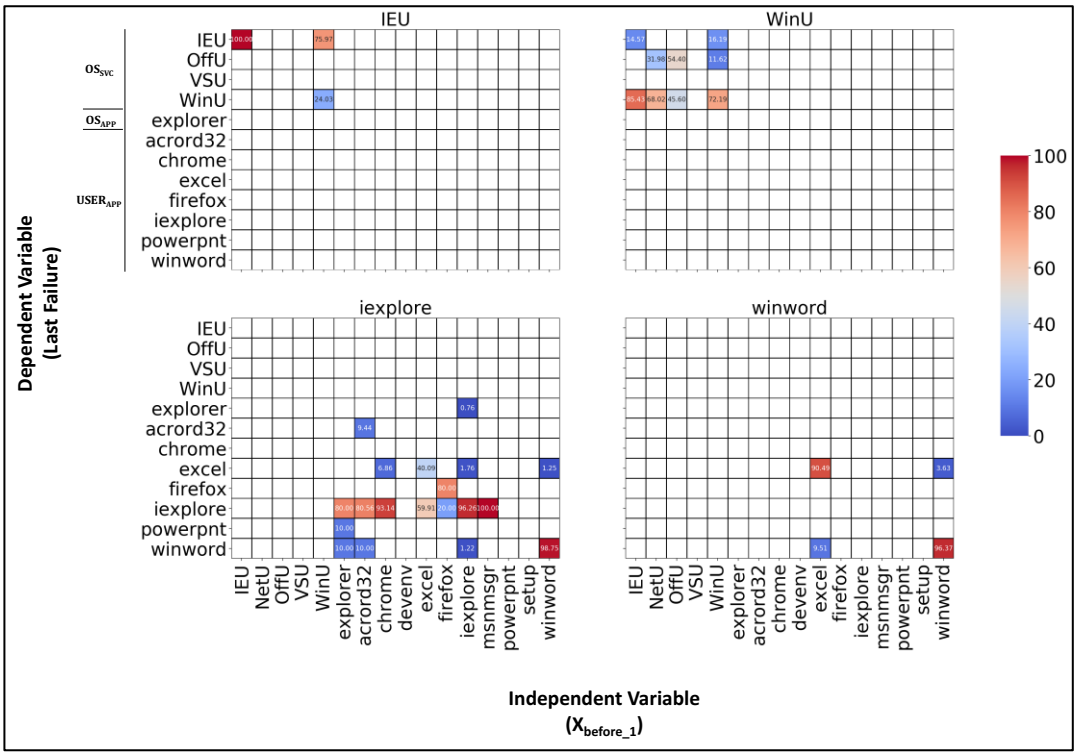


Figure 5.18. Probabilities based on sequences of three failure types (Random Forest - Filtered Sample_ALL).

Figures A.28 to A.30, in the Appendix, show the Decision Trees created from each sample. The long ramification to the left side is still present in all trees. Finally, the best number of trees in the Random Forest was 49 for Sample_OS, 88 for Sample_USER$_{APP}$, and 49 for Sample_ALL.

In general, all models presented good to high accuracies (86% to 93%) throughout all the prediction analyses performed. The Decision Tree models have the advantage of creating a tree that easily displays how the values are distributed in the samples. However, as the tree grows, its interpretability is impaired. The Random Forest is a black-box model that tries to mitigate the overfitting in Decision Trees, which occurs when the model has a great fit to the training data, but a poor fit to the test data. However, like the Decision Tree, it could not estimate reliable probabilities from unseen data during the training phase (see Section 5.4.2). On the other hand, the Multinomial Logistic Regression with Ridge regularization was capable of estimating the probabilities in such cases. Therefore, the Multinomial Logistic Regression with Ridge regularization is chosen to be used in future works to implement online failure prediction.

For example, as observed in Section 5.4.3, there is no occurrence in the training sample of DRIVER_POWER_STATE_FAILURE events occurring before any failure within the interval Minimum ≤ Δt ≤ Median. Therefore, both Decision Tree and Random Forest models could not calculate such probability. Consequently, if a DRIVER_POWER_STATE_FAILURE event occurs within interval Minimum ≤ Δt ≤ Median, the online failure prediction using the probabilities obtained by both Decision Tree and Random Forest models, would not able to provide any preventive measures.

6. CONCLUSION

6.1 Introduction

This chapter presents the final considerations of the empirical exploratory study conducted in this research.

Section 6.2 provides a summary of the empirical findings regarding the dependency of software failure events.

Section 6.3 highlights the main results and conclusion drawn by discovering patterns of multiple-event failures.

Section 6.4 highlights the main results and conclusion drawn by predicting failures based on patterns of failure sequences.

Section 6.5 presents the threats to the validity of the research.

Section 6.6 presents the contributions to the literature.

Section 6.7 presents the next steps planned for this research.

This research presents an exploratory study on multiple-event failure manifestations observed empirically in four different real workplaces and proposes a statistical approach to predict failures based on patterns of this type of failure manifestation. To have a complete comprehensive view of how multiple-event failures occur in the software layer of computer systems, I investigate the failure behavior of a commodity operating system (OS) and User Applications running on top of it.

Different from the traditional approach of analyzing OS failures that only considers failures that occurred in the Kernel space, I followed the method introduced in [29] that states that assessing an OS reliability by focusing on only Kernel-space failures is inaccurate since modern operating systems have several components running in both the Kernel and the User spaces. Consequently, the used method classifies the OS failures into three categories: OS Application (OS_{APP}), OS Service (OS_{SVC}), and OS Kernel (OS_{KNL}).

I highlight that the OS Kernel failures represented only 22.03% of all OS failures observed in the dataset, and the most prevalent OS failures were related to OS services (60.61%). Therefore, corroborating the study in [29], it can be seen that software reliability studies focused on operating systems should not restrict their analyses to only OS Kernel failures as has been the norm in the literature.

I also found that failures in the *Windows Update Service* (WUS) were predominant among the OS services failures, which means that the software update malfunctioning has a high impact on the OS reliability.

6.2 Dependence of Software Failure Events

A common characteristic among the majority of software reliability models is to assume that software failures occur independently [20]-[22]. This assumption is often used to adapt analytical models to mathematically treatable forms and simplify the estimation calculation of the model parameters [30]. However, this independence assumption may interfere, significantly, with the results of the software reliability assessment when the events of interest are not independent. Therefore, I tested the hypothesis of independence of failure events in real-world system failure data and found empirical evidence that the assumption of independence does not hold for many of the failures analyzed in this work. Figure 6.1 shows one of these real examples of failure dependence observed in the failure dataset analyzed. In this case, the failure of the RPC (Remote Procedure Call) server, which is used by the *Windows Update Service* and other running services in Windows 7, caused a failure in the *Windows Update Service* that, consequently, caused the failure of all updates being performed by the *Windows Update Service*.

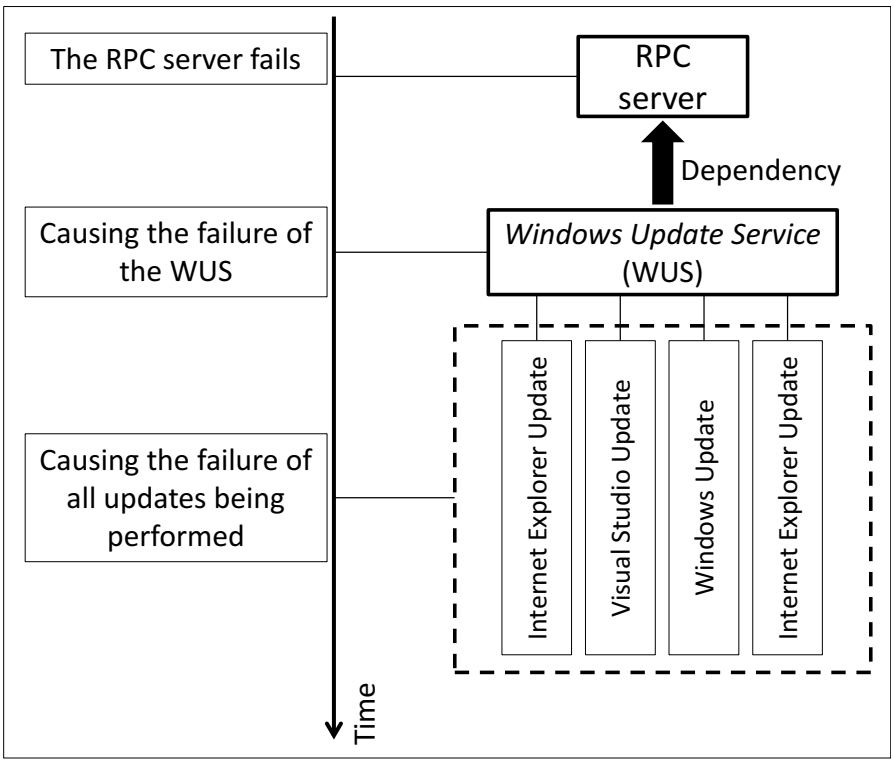


Figure 6.1. Example of software failures dependency.

6.3 Patterns of Multiple-Event Failures

This research investigates the dynamics of multiple-event failure manifestations. Therefore, the first question I wanted to answer was: Q1: Do software systems present systematic multiple-event failure manifestations?

To work through this question, I initially propose a method to discover patterns of failure associations (see Sections 4.2 and 5.2), which was applied to failures from a widely used commercial off-the-shelf Operating System (Sample_OS). As a result, I discovered 45 systematic OS failure associations with 153,511 occurrences, which are composed of the same or different failure types. Besides been the predominant OS failure in the dataset, failures in the *Windows Update Service* (WUS) are also prevalent in the associations. Most associations are composed of repetitions of WUS failures, which suggests that the *Windows Update Service* may have tried repeatedly and unsuccessfully to update the same component, which generated patterns of multiple same-type WUS failures. Moreover, associations composed of different WUS failures are also observed, suggesting that failures in the *Windows Update Service* could cause failures in all updates being performed by the service at the same time.

Answering Q1, the associations found indicate that most OS failures observed occur following a specific temporal order, which is robust evidence of systematic manifestations of multiple-event failures.

Multiple-event failures are characterized by sequences of failure events, varying in terms of length, duration, and combination of failure types. Therefore, two more questions were raised: Q2: Do software failures occur in a sequential pattern? If so, Q3: Do the observed sequential patterns reflect a causal relationship between the failures?

To answer these questions, I propose an improvement of the previous method by creating a protocol to discover patterns of failure sequences using varying time thresholds (see Sections 4.3 and 5.3). To provide a holistic view of multiple-event failures at the whole software layer, this method was applied to all three samples (see Sample_OS, Sample_$USER_{APP}$, and Sample_ALL in Section 3.4) analyzed in this work. As a result, I found 165 different OS failure sequences with 806 occurrences; 480 User Application failure sequences with 1,718 occurrences; and 640 sequences containing both OS and User Application failures with a total of 2,509 occurrences.

Hence, answering Q2, the sequences found are evidence of sequential patterns of different types of software failures.

Examining the failure sequences, I found that the most common sequence length is two, in any sample, which is the minimum number of failures that compose the sequences. Moreover, most sequences are composed of failures with the same type and cause. System-wide component failures, like the malfunctioning of frameworks and missing or corrupted system files, are the main cause of OS failures in the sequences analyzed (Section 5.3.1). Specifically, for User Application failures, problems related to memory management are their main cause. All of these failure causes are common problems that provoke multiple failures over time. Thus, proactive management of these factors (e.g., preventive maintenance of frameworks or check for missing files before performing update installation procedures) focused on their higher availability is a possible action to avoid failures in sequence and thus improve the system reliability.

Hence, answering Q3, based on the evidence obtained for the failures observed in the sequences, which were mainly caused by the above-mentioned system-wide component failures, it is not possible to confirm the existence of causal relationship among them.

I also investigate if OS and User Application failures may affect each other. Despite observing some occurrences of these failures together, they are rare (1.19% of the sequences found) and I could not determine how one affected the other. The more plausible explanation is the high frequency of some User Application failures in the dataset, which are so numerous that they occur together with some OS failures in different computer systems, and, consequently, are considered as associations/sequences by the proposed protocols. For example, the total amount of OS failures in the dataset is 7,010 while the total amount of failures of only the User application *iexplorer.exe* is 6,495 (see Table 3.5).

In summary, I found occurrences of associations and sequences in the failure dataset analyzed, which is empirical evidence that demonstrates the existence of well-established manifestations of multiple-event failures in a real software system. Lastly, this evidence raises the following question: Q4: Are these failure manifestations systematic enough to be considered genuine patterns?

Answering Q4, given the diversity of the groups evaluated, and the number of occurrences detected, such evidence strengthens my conclusion that the failure associations and sequences found are genuine patterns of multiple-event failures.

6.4 Prediction of Software Failures

To take advantage of the sequential nature of the multiple-event failures, as summarized in Section 6.3, the following questions were also addressed: Q5: Could we use these patterns to predict future failures? If so, Q6: At what level of accuracy? Q7: Would it be feasible to be implemented as part of an online failure prediction system?

Answering Q5, I used three methods to tackle the prediction problem; Multinomial Logistic Regression (w/ and w/o Ridge regularization), Decision Tree, and Random Forest. Based on failure sequences observed in all samples analyzed, the methods calculate the probability of a certain failure event to occur within a time interval upon the occurrence of a particular pattern of preceding failures. In other words, these methods can predict the most probable failure to occur (i.e., the most probable future failure) based on patterns of the preceding failures.

Answering Q6, the proposed prediction approach was able to estimate models to predict software failures with good to high accuracy (86% to 93%), considering different input data such as categories, types of failures, and Δt between failures.

I observed that failures of the same type are more probable of occurring together and this pattern was observed throughout the prediction analysis in all samples analyzed. Moreover, the resulting models showed robust enough to deal with different prediction time intervals, varying the time between failures from seconds to hours with no degrading effect on their accuracy.

Answering Q7, the empirical evidence observed in this research seems promising and suggests the feasibility of using the proposed approach to implement online failure prediction, which must be composed of two main tasks. One task is related to continuously calibrating the model's parameters with new incoming failure records, and the other task is performing the probability estimation of the next failures to occur, as well as the appropriate preventive actions based on these outputs (e.g., reinstall a DLL file upon its failed update). The calibrating task can be performed either offline or online, and the prediction task must be done online.

6.5 Threats to Validity

Like any empirical research work, this study has limitations that must be considered when interpreting its results. In this subsection, I highlight these validity threats [94] and the strategies adopted to mitigate them.

6.5.1 Internal Validity

This validity addresses the quality of evidence adopted to support my study's conclusions.

Since this study is an observational study rather than a controlled study, the failures analyzed here were collected from computers in real production environments and not obtained by controlled experiments. Consequently, my conclusions are drawn from findings obtained from associations and sequences of observed failure events and not based on the direct manipulation of the system under study to produce the failure events analyzed.

We know from the literature (e.g., [94], [95]) that controlled experiments offer a higher internal validity than observational ones. On the other hand, they impose important limitations in terms of external validity. To achieve a balance between both validities, I chose the observational approach and worked with a dataset composed of failures from different populations (or groups). Based on this approach, I aim at mitigating mainly validity threats such as systematic errors and selection bias.

In this study, systematic errors could be caused by environmental factors, such as the same administrative policy and OS settings applied to the entire group of computers. The proposed methods were designed to mitigate these threats, considering as failure patterns only the combinations of failures observed consistently in multiple groups of computers.

Selection bias is another threat to internal validity that arises in this study, which was also mitigated to the extent that we collected the failure samples from various computers (644 in total) of dissimilar workplaces. Three (G1, G2, and G4) out of four workplaces were under different administration policies and they were all under varying workloads, which intentionally aimed at greater diversity in terms of possible failure scenarios covered by the dataset.

Another important aspect related to internal validity that must be considered is related to the failure causes discussed in this work. This information was obtained through the *Error Codes*, *Exception Codes,* and *Stop Codes*. The assignment of these codes to the failure records was performed by Win7 itself, at the failure time; thus, their correctness is assumed to be right. My understanding when learning about the *Codes* was based on official Microsoft technical documentation, which is always susceptible to misinterpretation. However, given that the description of most *Codes* used in this study was straightforward, I consider this issue of minor threat.

The evaluation of the temporal occurrence of the failures was based on the RAC's record field, *TimeGenerated*, also created by Win7 automatically at the failure time. It is important to highlight that this field stores the time in seconds, so two failures that occur consecutively within a time interval of less than one second are registered with the same timestamp value. In this case, these two failures would be considered erroneously, as occurring at the same time. Therefore, the Δt between them is equal to zero. This is a time precision limitation of Win7's RAC instrumentation. However, part of the percentage of Δts equal to zero may represent failure events that correctly occurred at the same time.

6.5.2 External Validity

This validity addresses the extent to which my results can be generalized. First, it is important to highlight that all empirical results presented in this study apply to Windows 7 operating systems. As mentioned in Section 3.2, my choice for this specific OS is due to the fact that this is one of the most widely used mass-market OS nowadays. Thus, although limited to this OS platform, it is expected that the applicability of my empirical results is wide-spread.

Hence, if one wants to generalize this work's empirical results to another population of Win7 machines, then I consider that the external validity of this study is high, especially due to the size and diversity of the analyzed dataset. Though the reported empirical results are based on Win7, I conjecture that similar findings may be found investigating other operating systems from the Windows OS family that are architecturally close to Win7, such as XP, Vista, Win8/8.1, and Win10.

In addition to the empirical results, it is important to consider the generalizability of the methods proposed. The methods are general enough to be applied to failure data from other software systems if their failure records contain at least the following information: *failed software identification* (Failure category and/or type/subtype), and *failure time stamp*. The majority of software systems keep this information in their event logs.

6.6 Contributions to the literature

The results of this work were the basis for writing three papers. The first, entitled, "Failure Patterns in Operating Systems: An Exploratory and Observational Study" [30], is published in the 2018 edition of the Journal of Systems and

Software. It describes in detail the protocol developed for discovering failure association patterns.

The second, entitled “An Empirical Exploratory Analysis of Failure Sequences in a Commodity Operating System” [14], is published in the Proceedings of the 2019 edition of the Brazilian Symposium on Computing Systems Engineering. The paper won the best paper award of the symposium. It presents the protocol developed to discover patterns of failure sequences.

Finally, the third paper, entitled “A Statistical Approach to Predict Operating System Failures Based on Multiple Failures Association” [75], is published in the Proceedings of the 2020 edition of the Brazilian Symposium on Computing Systems Engineering.

6.7 Future Work

Based on the results of this research, we are working on two ideas. The first, as mentioned in Section 6.4, is to implement an online failure prediction analysis based on the probabilities obtained in the prediction analysis. The second is to replicate this study using a dataset based on failure records from Win10. Therefore, the main idea is to test the hypothesis of similarity in failure patterns in both operating systems. Besides, we will test other prediction methods (e.g., Neural Networks) and collect complete logs from the system, not just failure logs. Consequently, other variables will be considered in the prediction models, like the workload in progress when the failure occurred, for example.

REFERENCES

[1] C. Jee and T. Macaulay, “Top software failures in recent history,” https://www.computerworld.com/article/3412197/top-software-failures-in-recent-history.html (Accessed: January 20, 2021).

[2] R. Charette, “The Biggest IT Failures of 2018,” https://spectrum.ieee.org/riskfactor/computing/it/it-failures-2018-all-the-old-familiar-faces (Accessed: January 20, 2021).

[3] N. G. Leveson and C. S. Turner, “An Investigation of the Therac-25 Accidents”, *IEE Computer*, vol. 26, no.7, pp.18-41. DOI: https://doi.org/10.1109/MC.1993.274940

[4] M. Dowson, "The Ariane 5 software failure," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 2, pp. 84, 1997. DOI: https://doi.org/10.1145/251880.251992

[5] J. J. Angel, "When finance meets physics: Impact of the speed of light on financial markets and their regulation," *The Financial Review*, vol. 49, no. 2, pp. 271-281, 2014. DOI: https://doi.org/10.1111/fire.12035

[6] A. Levin, “Latest 737 Max Fault That Alarmed Test Pilots Rooted in Software,” https://www.bloomberg.com/news/articles/2019-07-27/latest-737-max-fault-that-alarmed-test-pilots-rooted-in-software (Accessed: January 20, 2021).

[7] R. Wall and M. Sherman, “The Multiple Problems, and Potential Fixes, With the Boeing 737 MAX,” https://www.wsj.com/articles/fixing-the-problems-with-boeings-737-max-11566224866 (Accessed: January 20, 2021).

[8] P.L. Li, M. Ni, S. Xue, J.P. Mullally, M. Garzia, and M. Khambatti, "Reliability Assessment of Mass-Market Software: Insights from Windows Vista," in Proceedings of the 19th International Symposium on Software Reliability Engineering, 2008, pp. 265-270. DOI: https://doi.org/10.1109/issre.2008.60

[9] G. Bruzzone, M. Caccia, A. Bertone, and G. Ravera, “Standard Linux for Embedded Real-Time Robotics and Manufacturing Control Systems”, in Proceedings of the 14th Mediterranean Conference on Control and Automation, 2006, pp. 01-06. DOI: https://doi.org/10.1109/med.2006.328773

[10] D. Abbott, “Linux for Embedded and Real-time Applications,” Newnes, 3ª ed., 296 pgs, 2012.

[11] F. Salfner, M. Schieschke, and M. Malek, "Predicting failures of computer systems: a case study for a telecommunication system," in Proceedings of the 20th International Conference on Parallel and distributed processing, 2006, pp. 348-348. DOI: https://doi.org/10.1109/ipdps.2006.1639672

[12] M. Lyu, "Software Reliability Engineering: A Roadmap," in Proceedings of the 29th International Conference Software Engineering, Future Software Engineering, 2007, pp. 153 -170. DOI: https://doi.org/10.1109/fose.2007.24

[13] ANSI/IEEE, Standard Glossary of Software Engineering Terminology, STD-729-1991, 1991.

[14] C.A.R. Dos Santos, R. Matias, and K. S. Trivedi, “An Empirical Exploratory Analysis of Failure Sequences in a Commodity Operating System,” in Proceedings of the Brazilian Symposium on Computing Systems Engineering, 2019. DOI: https://doi.org/10.1109/sbesc49506.2019.9046072

[15] B. Schroeder and G. Gibson, “A large-scale study of failures in high-performance computing systems,” in Proceedings of the International Conference on Dependable Systems and Networks, 2006, pp.249-258. DOI: https://doi.org/10.1109/dsn.2006.5

[16] A. Ganapathi, V. Ganapathi and D. Patterson, “Windows XP kernel crash analysis,” in Proceedings of the Conference on Large Installation System Administration, 2006, pp. 149-159.

[17] M. M. Swift, B. N. Bershad and H. M. Levy, “Improving the reliability of commodity operating systems,” in Proceedings of the ACM Symposium on Operating Systems Principles, 2003, pp.207-222. DOI: https://doi.org/10.1145/945445.945466

[18] G. J. Pai "A Survey of Software Reliability Models". Technical Report CS 651: Dependable Computing. Department of ECE, University of Virginia, 2002.

[19] K. Goseva-Popstojanova and K. S. Trivedi, "Architecture Based Software Reliability Prediction," *Computers & Mathematics with Applications*, vol. 46, no. 7, pp. 1023-1036, 2003. DOI: https://doi.org/10.1016/S0898-1221(03)90116-7

[20] K. Goseva-Popstojanova and K. S. Trivedi, "The Effects of Failure Correlation on Software Reliability and Performability," in Proceedings of the 29th International Symposium Fault Tolerant Computing, 1999, pp. 45-46.

[21] K. Goseva-Popstojanova and K. S. Trivedi, "Effects of Failure Correlation on Software in Operation," in Proceedings of the Pacific Rim International Symposium on Dependable Computing, 2000, pp. 69-76. DOI: https://doi.org/10.1109/prdc.2000.897286

[22] K. Goseva-Popstojanova and K. S. Trivedi, "Failure Correlation in Software Reliability Models," *IEEE Trans. On Reliability*, vol.49, no.1, pp. 37-48, 2000. DOI: https://doi.org/10.1109/24.855535

[23] C. Dyer and M. Taylor, “Cot deaths: law in disarray as 258 cases of convicted parents to be reviewed,” https://www.theguardian.com/society/ 2004/jan/20/health.childprotection (Accessed: January 20, 2021).

[24] R. Hill, "Reflections on the cot death cases," *Significance*, 2, pp. 13-15, 2005. DOI: https://doi.org/10.1111/j.1740-9713.2005.00077.x

[25] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11-33, 2004. DOI: https://doi.org/10.1109/TDSC.2004.2

[26] E. E. Lewis, “Introduction to Reliability Engineering,” Wiley Publishing, 2ª ed., 464 pgs, 1995.

[27] A. Ganapathi and D. Patterson, “Crash Data Collection: A Windows Case Study,” in Proceedings of the International Conference on Dependable Systems and Networks, 2005, pp. 280-285. DOI: https://doi.org/10.1109/dsn.2005.32

[28] Y. S. Dai , M. Xie, and K. L. Poh, “Modeling and Analysis of Correlated Software Failures of Multiple Types, ” *IEEE Transactions on Reliability*, vol. 54, no. 1), pp. 100-106, 2005. DOI: https://doi.org/10.1109/tr.2004.841709

[29] R. Matias, G. Oliveira, L. Araujo, "Operating system reliability from the quality of experience viewpoint: an exploratory study," in Proceedings of the 28th ACM Symposium on Applied Computing, 2013, pp. 1644–1649. DOI: https://doi.org/10.1145/2480362.2480669

[30] C.A.R. Dos Santos and R. Matias, “Failure Patterns in Operating Systems: An Exploratory and Observational Study,” *Elsevier Journal of Systems and Software*, vol. 137, pp. 512-530, March 2018. DOI: https://doi.org/10.1016/j.jss.2017.03.058

[31] M. Kalyanakrishnam, Z. Kalbarczyk, R. Iyer, "Failure data analysis of a LAN of Windows NT based computers,"in Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems, 1999, pp. 178–187. DOI: https://doi.org/10.1109/reldis.1999.805094

[32] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in Proceedings of the 18th ACM Symposium on Operating Systems Principles, 2001, pp. 73–88. DOI: https://doi.org/10.1145/502059.502042

[33] R. Lai, “Huawei reveals HarmonyOS, its alternative to Android,” https://www.engadget.com/2019/08/09/huawei-harmony-os-hongmeng-android/ (Accessed: December 27, 2019).

[34] M. Golemati, A. Katifori, E. Giannopoulou, I. Daradimos, and C. Vassilakis, "Evaluating the significance of the Windows Explorer visualization in personal information management browsing tasks," in Proceedings of the International Conference on Information Visualization, 2007, pp. 93-100. DOI: https://doi.org/10.1109/iv.2007.46

[35] C.A.R. Dos Santos, M. Antunes, R. Matias, L. Assunção, and V. Maciel, "Reliability Assessment of Commercial Off-the-Shelf Operating System Software: An Empirical Study," in Proceedings of the Brazilian Symposium on Computing Systems Engineering, 2018. DOI: https://doi.org/10.1109/sbesc.2018.00038

[36] Microsoft, "Microsoft Security Essentials," https://www.microsoft.com/en-us/download/details.aspx?id=5201 (Accessed: December 27, 2019).

[37] Microsoft, "Windows Error Reporting," http://msdn.microsoft.com/en-us/library/bb513613(v=vs.85).aspx (Accessed: December 27, 2019).

[38] Microsoft, "Windows Installer," http://msdn.microsoft.com/en-us/library/cc185688%28VS. 85%29.aspx (Accessed: December 27, 2019).

[39] D. Hosmer, S. Lemeshow, and R. Sturdivant, "Applied Logistic Regression," 3rd ed., John Wiley & Sons, 500 pgs, 2013.

[40] I. H. Witten, E. Frank, and M. A. Hall, "Data Mining: Practical Machine Learning Tools and Techniques," Morgan Kaufmann, 664 pgs, 2011.

[41] P. Bühlmann and S. van de Geer, "Statistics for High-Dimensional Data: Methods, Theory and Applications," Springer, 558 pgs, 2011.

[42] T. Hastie, R. Tibshirani, and J. Friedman" The Elements of Statistical Learning: Data Mining, Inference, and Prediction," Springer, 745 pgs, 2019.

[43] G. Tutz and J. Gertheiss, "Regularized regression for categorical data," *Statistical Modelling*, vol. 16, no.3, pp.161-200, 2016. DOI: https://doi.org/10.1177/1471082x16642560

[44] A. E. Hoerl and R. W. Kennard, "Ridge Regression: Biased Estimation for Nonorthogonal Problems," *Technometrics*, vol. 12, no.1, pp.55-67, 1970. DOI: https://doi.org/10.1080/00401706.1970.10488634

[45] J. Friedman, T. Hastie , and R. Tibshirani, "Regularization Paths for Generalized Linear Models via Coordinate Descent," *Journal of Statistical Software*, vol. 33, no.1, pp.01-22, 2010. DOI: https://doi.org/10.18637/jss.v033.i01

[46] F. Gorunescu "Data Mining: Concepts, Models and Techniques," Springer, 372 pgs, 2011.

[47] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone "Classification and Regression Trees," CRC Press, 358 pgs, 1984.

[48] D. Esposito and F. Esposito "Introducing Machine Learning," Microsoft Press, 400 pgs, 2020.

[49] M. Kantardzic "Data Mining: Concepts, Models, Methods, and Algorithms," Wiley-IEEE Press, 672 pgs, 2020.

[50] J. Han, M. Kamber, and J. Pei "Data Mining: Concepts and Techniques," Morgan Kaufmann, 703 pgs, 2011.

[51] T. Oshiro, P. Perez, and J. Baranauskas " How Many Trees in a Random Forest?," Lecture notes in computer science, 2012.

[52] J. A. Rosenthal, "Statistics and Data Interpretation for Social Work," Springer, 506 pgs, 2011.

[53] H-J Andreß, K. Golsch, and A. Schmidt, "Applied Panel Data Analysis for Economic and Social Surveys," Springer, 344 pgs, 2013.

[54] K. Pituch and J. Stevens, "Applied Multivariate Statistics for the Social Sciences: Analyses with SAS and SPSS," 6th ed., Routledge, 814 pgs, 2015.

[55] D. McFadden, “Conditional logit analysis of qualitative choice behavior,” In Frontiers in Econometrics, P. Zarembka (ed.), 1974, pp 105-142.

[56] R. Matias, M. Prince, L. Borges, C. Sousa, and L. Henrique, “An Empirical Exploratory Study on Operating System Reliability,” In Proceedings 29th ACM Symposium on Applied Computing, 2014, pp. 1523–1528. DOI: https://doi.org/doi.org/10.1145/2554850.2555021

[57] K. S. Trivedi, "Probability and Statistics with Reliability, Queuing, and Computer Science Applications," 2nd ed., vol. 1. Wiley, 830 pgs, 2001.

[58] C. Bird, V.P. Ranganath, T. Zimmermann, N. Nagappan, and A. Zeller. “Extrinsic Influence Factors in Software Reliability: A Study of 200,000 Windows Machines,” In Proceedings 36th International Conference on Software Engineering, 2014, pp. 205-214. DOI: https://doi.org/10.1145/2591062.2591173

[59] P. Fournier-Viger, J. Lin, R. Kiran, Y. Koh, and R. Thomas, “A Survey of Sequential Pattern Mining”, *Data Science and Pattern Recognition*, vol. 1, no.1, pp.54-77.

[60] R. Srikant, and R. Agrawal, “Mining sequential patterns: Generalizations and performance improvements,” Proceedings of the 5th International Conference on Extending Database Technology, 1996, pp. 3–17. DOI: https://doi.org/10.1007/BFb0014140

[61] M. J. Zaki, “SPADE: An efficient algorithm for mining frequent sequences,” *Machine learning*, vol. 42, no.1, pp.31-60. DOI: https://doi.org/10.1023/A:1007652502315

[62] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M. C. Hsu, “Mining sequential patterns by pattern-growth: The prefixspan

approach," *IEEE Transactions on knowledge and data engineering*, vol. 16, no.11, pp.1424-1440. DOI: https://doi.org/10.1109/TKDE.2004.77

[63] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential pattern mining using a bitmap representation," Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2002, pp. 429–435. DOI: https://doi.org/10.1145/775047.775109

[64] Z. Yang, and M. Kitsuregawa, "LAPIN-SPAM: An improved algorithm for mining sequential pattern," Proceedings of the 21st International Conference on Data Engineering Workshops, 2005, pp. 1222–1222. DOI: https://doi.org/10.1109/ICDE.2005.235

[65] M. J, Zaki and W. Meira, "Data Mining and Machine Learning: Fundamental Concepts and Algorithms," Cambridge University Press, 775 pgs, 2020.

[66] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo, "BlueGene/L Failure Analysis and Prediction Models," In Proceedings International Conference on Dependable Systems and Networks, 2006, pp. 425–434. DOI: https://doi.org/10.1109/DSN.2006.18

[67] Fronza, A. Sillitti, G. Succi, M. Terho, J. Vlasenko, "Failure Prediction Based on Log Files Using Random Indexing and Support Vector Machines", *Journal of Systems and Software*, vol. 86, no. 1, pp. 2-11, 2013. DOI: https://doi.org/10.1016/j.jss.2012.06.025

[68] NetMarketShare, "Desktop Operating System Market Share," https://netmarketshare.com/operating-system-market-share.aspx?id=platformsDesktopVersions (Accessed: January 20, 2021).

[69] Microsoft, "Reliability analysis component," http://technet.microsoft.com/en-us/library/cc774636(v=ws.10).aspx (Accessed: January 20, 2021).

[70] Microsoft, "Win_32 ReliabilityRecord class," https://msdn.microsoft.com/en-us/library/windows/desktop/ee706630%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396 (Accessed: January 20, 2021).

[71] Microsoft, "Understanding the System Stability Index," https://technet.microsoft.com/en-us/library/cc749032(v=ws.10).aspx (Accessed: January 20, 2021).

[72] HPDCS, "Survey on OS Failures," http://hpdcs.facom.ufu.br/osr-team/index.php (Accessed: January 20, 2021).

[73] M. Russinovich, D. A. Solomon, and A. Ionescu, "Microsoft Windows Internals," 7th ed., Microsoft Press, 900 pgs, 2009.

[74] Microsoft, "Did system uptime error cause chkdisk," https://answers.microsoft.com/en-us/windows/forum/all/did-system-uptime-error-cause-chkdisk/8d55525c-9f5b-4278-bd87-1598bce009ee (Accessed: January 20, 2021).

[75] C.A.R. Dos Santos, R. Matias, and K. S. Trivedi, “A Statistical Approach to Predict Operating System Failures Based on Multiple Failures Association,” in Proceedings of the Brazilian Symposium on Computing Systems Engineering, 2020, pp. 1–8. DOI: https://doi.org/10.1109/SBESC51047.2020.9277859

[76] E. Volna, M. Kotyrba, and M. Janosek, "Pattern recognition and classification in time series data," Advances in Computational Intelligence and Robotics, 1st ed. IGI Global, 282 pgs, 2016.

[77] F.H. Marriott, "A Dictionary of Statistical Terms," Longman Scientific & Technical/Wiley, 223 pgs, 1990.

[78] M. Scholes, and J. Williams, "Estimating betas from nonsynchronous data," *Journal of Financial Economics*, vol 5, no. 3, pp. 309–327, 1977. DOI: https://doi.org/10.1016/0304-405X(77)90041-1

[79] M.C. Lundin, M.M. Dacorogna, and U. A. Müller, "Correlation of high frequency financial time series". Lequex, P. (Ed.), The Financial Markets Tick by Tick, Chapter 4. John Wiley & Sons, pp. 91-126, 1999.

[80] T. Hayashi, N. Yoshida, "On covariance estimation of non-synchronously observed diffusion processes", *Bernoulli*, vol 11, no. 2 , pp. 359–379, 2005. DOI: https://doi.org/10.3150/bj/1116340299

[81] K. Rehfeld, N. Marwan, J. Heitzig, and J.Kurths, "Comparison of correlation analy- sis techniques for irregularly sampled time series," *Nonlinear Processes Geophys*, vol. 18, no. 3, pp. 389-404, 2011. DOI: https://doi.org/10.5194/npg-18-389-2011

[82] A. Eckner, “A framework for the analysis of unevenly-spaced time series data,” http://eckner.com/papers/unevenly_spaced_time_series_analysis.pdf (Accessed: January 20, 2021).

[83] B. Desharnais, F. Camirand-Lemyre, P. Mireault, and C.Skinner, “Determination of confidence intervals in non-normal data: application of the bootstrap to cocaine concentration in femoral blood,” *Journal of analytical toxicology*, vol. 39, pp. 113-117, 2015. DOI: https://doi.org/10.1093/jat/bku127

[84] T. Beauvisage, "Computer usage in daily life," in Proceedings of International Conference on Human Factors in Computing Systems, 2009, pp. 575-584. DOI: https://doi.org/10.1145/1518701.1518791

[85] Microsoft, “Windows update showing error 80070643,” http://answers.microsoft.com/en-us/insider/forum/insider_wintp-insider_update/windows-update-showing-error-80070643/0e53bf0f-8843-45a1-b3c4-0940c516c8d9 (Accessed: January 20, 2021).

[86] Microsoft, “Troubleshooting Issues When You Use the Discovery Wizard to Install an Agent,” https://docs.microsoft.com/en-us/previous-versions/system-center/operations-manager-2007-

r2/ff358634(v=technet.10)?redirectedfrom=MSDN (Accessed: January 20, 2021).

[87] Microsoft, “Windows 7 Update Problems - Code 800B0100,” https://answers.microsoft.com/en-us/windows/forum/all/windows-7-update-problems-code-800b0100/c2d0f18b-dbef-455d-a32e-b730ebd2370f (Accessed: January 20, 2021).

[88] Microsoft, “Windows Update Agent - Error Codes,” http://social.technet.microsoft.com/wiki/contents/articles/15260.windows-update-agent-error-codes. aspx . (Accessed: January 20, 2021).

[89] Microsoft, “NTSTATUS Values,” https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/596a1078-e883-4972-9bbc-49e60bebca55?redirectedfrom=MSDN (Accessed: January 20, 2021).

[90] Microsoft, “NTSTATUS 0xc000041d,” https://support.microsoft.com/en-us/help/2856748/ntstatus-0xc000041d-error-occurs-when-you-click-a-link-in-a-web-based (Accessed: January 20, 2021).

[91] Microsoft, “Windows Update error 80242016,” https://docs.microsoft.com/en-us/windows/deployment/update/windows-update-error-reference (Accessed: January 20, 2021).

[92] Microsoft, “Add-WindowsFeature sometimes fails with error 0x800f0902,” https://social.technet.microsoft.com/Forums/lync/en-US/1eb8c5c2-64bf-4d07-8020-dac902230688/addwindowsfeature-sometimes-fails-with-error-0x800f0902-the-operation-cannot-be-completed-because?forum=winservermanager (Accessed: January 20, 2021).

[93] M. Crowley, "Pro Internet Explorer 8 & 9 Development: Developing Powerful Applications for The Next Generation of IE," Apress, 446 pgs, 2010.

[94] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering," In Proceedings 37th International Conference on Software Engineering, 2015, pp. 9-19. DOI: https://doi.org/10.1109/ICSE.2015.24

[95] R. Feldt and A. Magazinius, "Validity threats in empirical software engineering research - An initial survey, " in Proceedings 22nd International Conference on Software Engineering and Knowledge Engineering, 2010, pp. 374-379.

APPENDIX

Table A.1. Reference Failures that appeared in four groups.

Category	OS Failure Type [Subtype]	G1	G2	G3	G4	Total
OS_{SVC}	WUS [Windows Update]	23	613	5	523	1,164
OS_{APP}	explorer.exe	35	226	29	296	586
OS_{SVC}	WUS [.Net Framework Update]	3	187	3	116	309
OS_{SVC}	WUS [Office Update]	44	87	1	134	266
OS_{KNL}	Windows-StartupRepair	66	135	2	34	237
OS_{SVC}	msiexec.exe	31	3	3	23	60
OS_{SVC}	dllhost.exe	2	28	13	4	47
OS_{KNL}	SYSTEM_THREAD_ EXCEPTION_NOT HANDLED	1	30	7	3	41
OS_{SVC}	services.exe	1	16	5	19	41
OS_{APP}	mmc.exe	3	10	4	7	24
OS_{SVC}	taskhost.exe	1	1	1	1	4

Table A.2. Reference Failures that appeared in three groups.

Category	OS Failure Type [Subtype]	G1	G2	G3	G4	Total
OS_{SVC}	WUS [Internet Explorer Update]	7	1,461	-	118	1,586
OS_{KNL}	VIDEO_TDR_ERROR	825	62	-	11	898
OS_{APP}	mscorsvw.exe	-	115	6	159	280
OS_{SVC}	WUS [Visual Studio Update]	-	55	103	41	199
OS_{APP}	rundll32.exe	-	19	7	155	181
OS_{KNL}	DRIVER_POWER _STATE_FAILURE	-	22	23	78	123
OS_{SVC}	svchost.exe	-	28	24	53	105
OS_{SVC}	WUS [MS C++ Update]	-	16	5	19	40
OS_{SVC}	WUS [XML Update]	-	16	1	16	33
OS_{KNL}	CRITICAL_ OBJECT_TERMINATION	18	4	-	6	28
OS_{KNL}	MEMORY _MANAGEMENT	-	22	1	2	25
OS_{KNL}	PAGE_FAULT_IN_NONPAGED_AREA	-	7	1	10	18
OS_{KNL}	DRIVER_IRQL_ NOT_LESS_OR_EQUAL	1	11	-	5	17
OS_{SVC}	WUS [MS Silverlight Update]	-	4	2	11	17
OS_{APP}	dwm.exe	4	-	1	4	9
OS_{SVC}	WUS [Visio Viewer Update]	1	5	-	2	8
OS_{KNL}	CLOCK_WATCHDOG _TIMEOUT	1	3	-	1	5

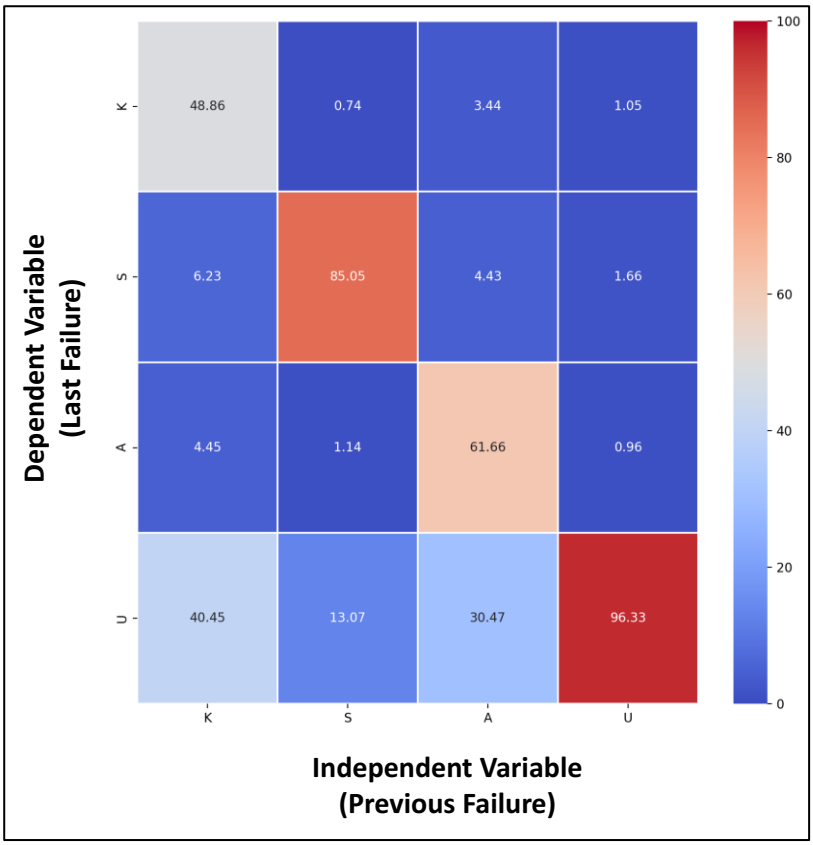


Figure A.1. Probabilities based on sequences of two failure categories (Random Forest).

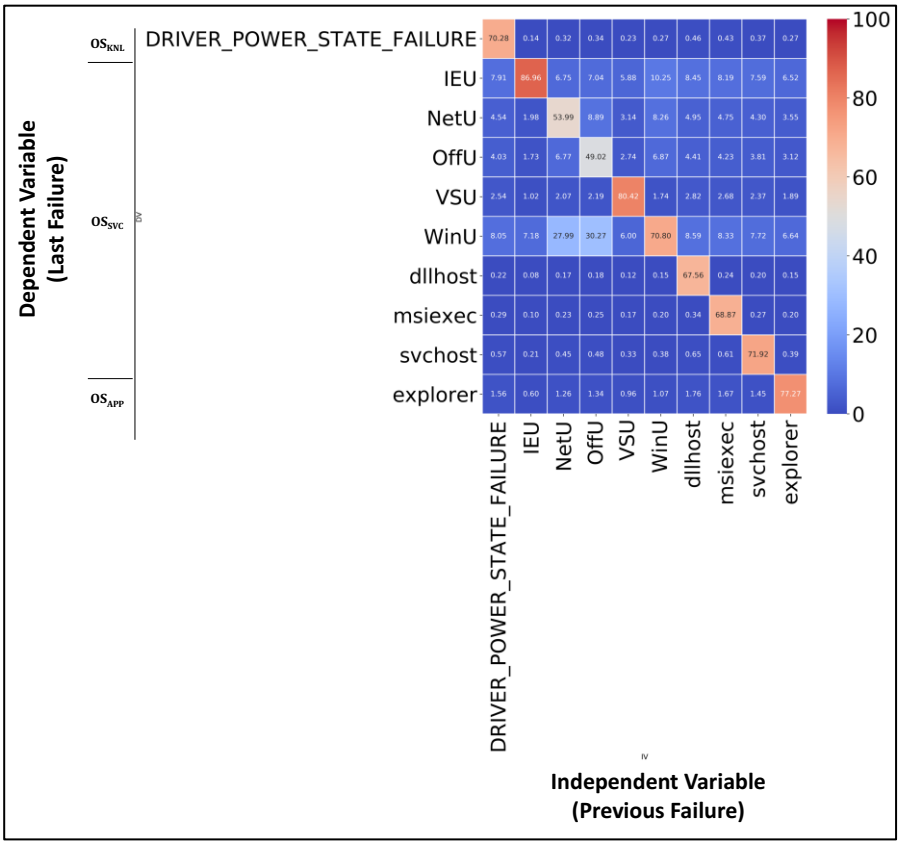


Figure A.2. Probabilities based on sequences of two failure types (Multinomial Logistic Regression with Ridge - Sample_OS).

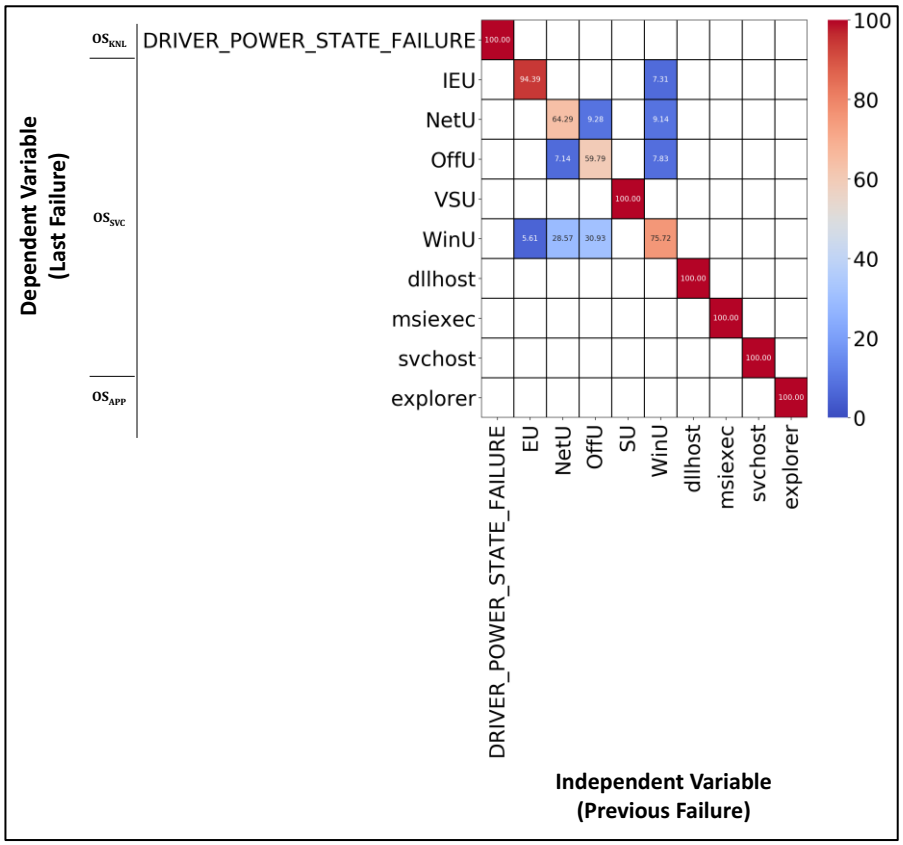


Figure A.3. Probabilities based on sequences of two failure types (Decision Tree - Sample_OS).

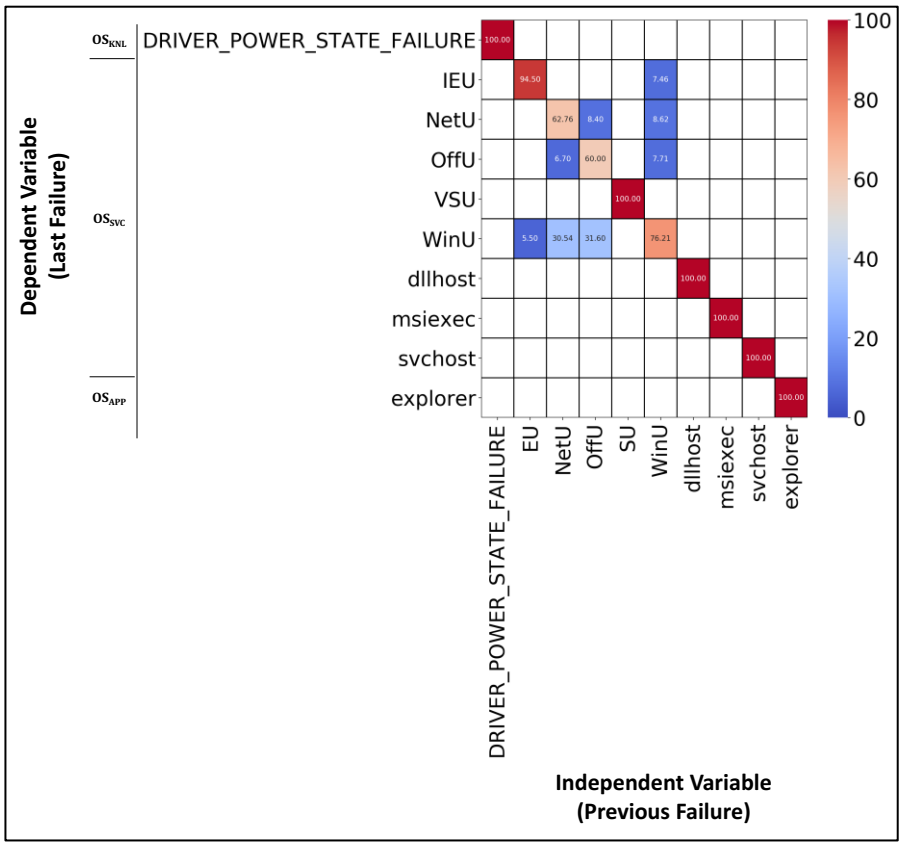


Figure A.4. Probabilities based on sequences of two failure types (Random Forest - Sample_OS).

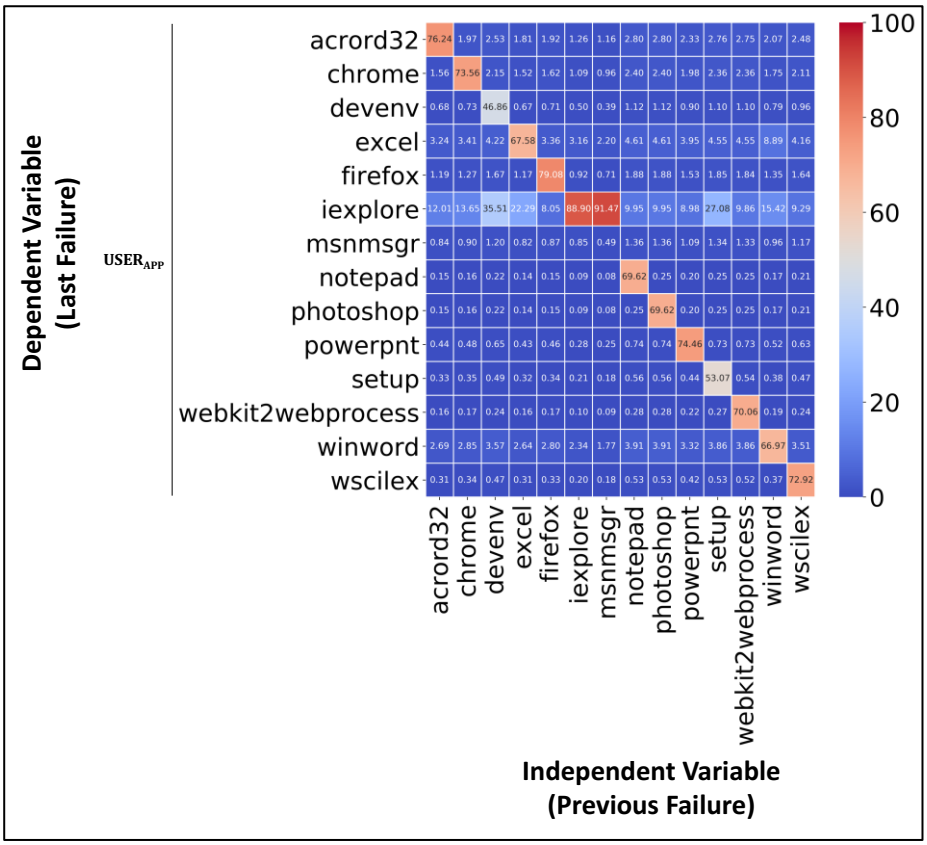


Figure A.5. Probabilities based on sequences of two failure types (Multinomial Logistic Regression with Ridge - Sample_USER$_{APP}$).

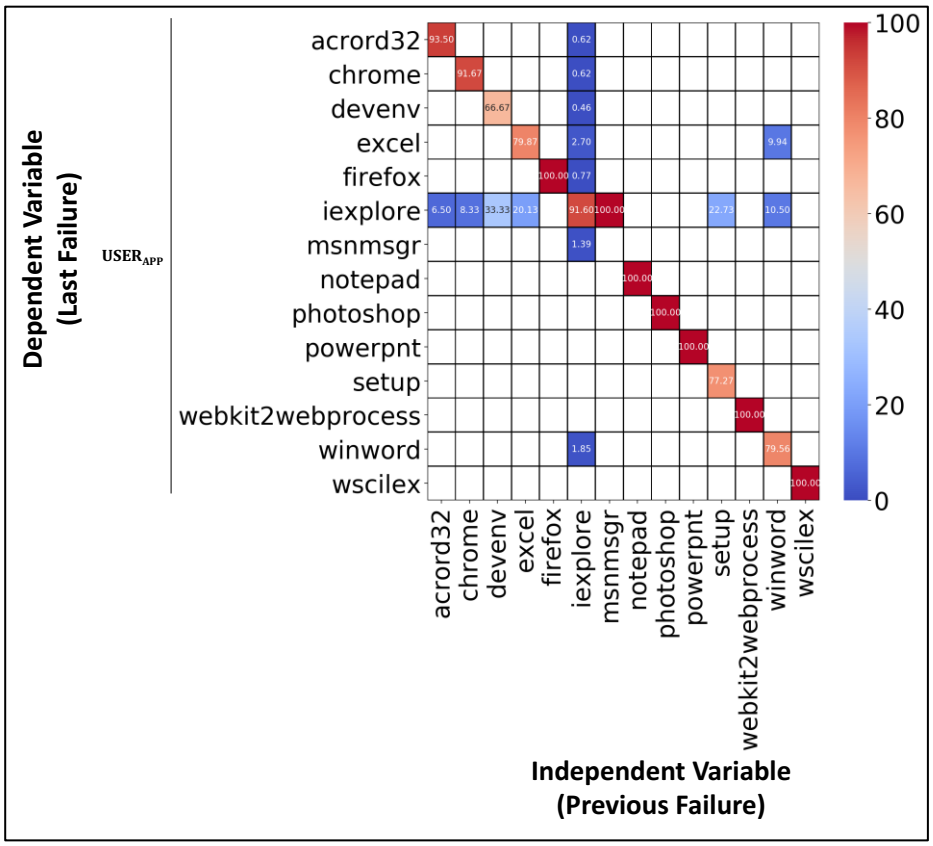


Figure A.6. Probabilities based on sequences of two failure types (Decision Tree - Sample_USER$_{APP}$).

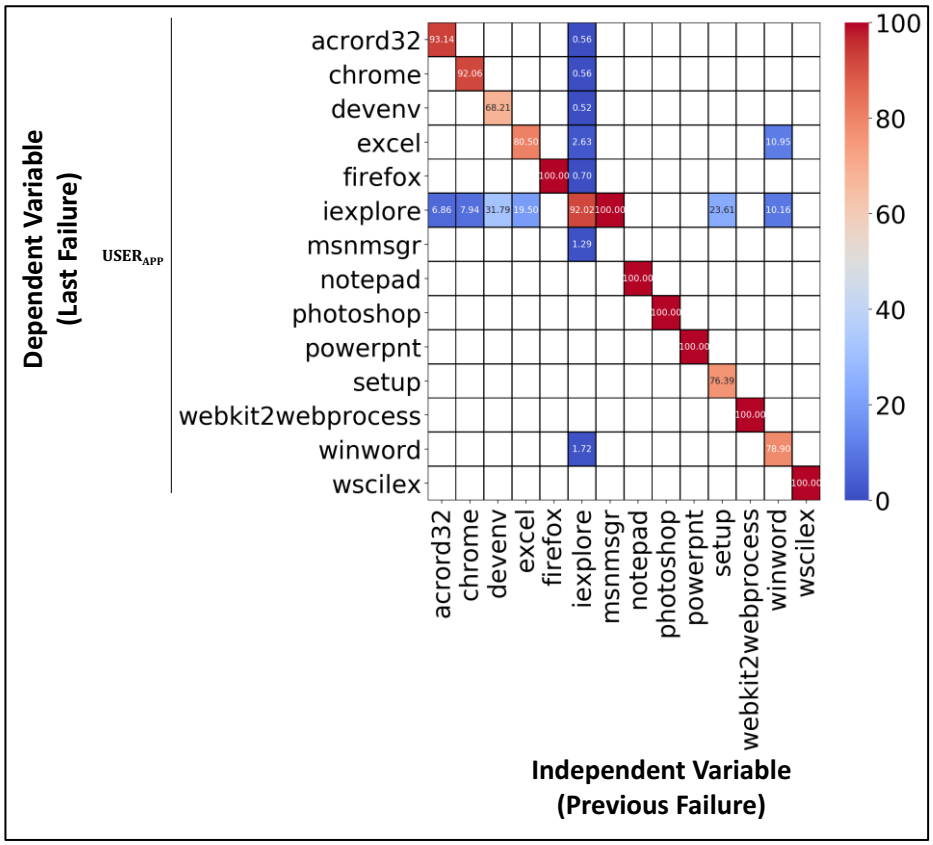


Figure A.7. Probabilities based on sequences of two failure types (Random Forest - Sample_USER$_{APP}$).

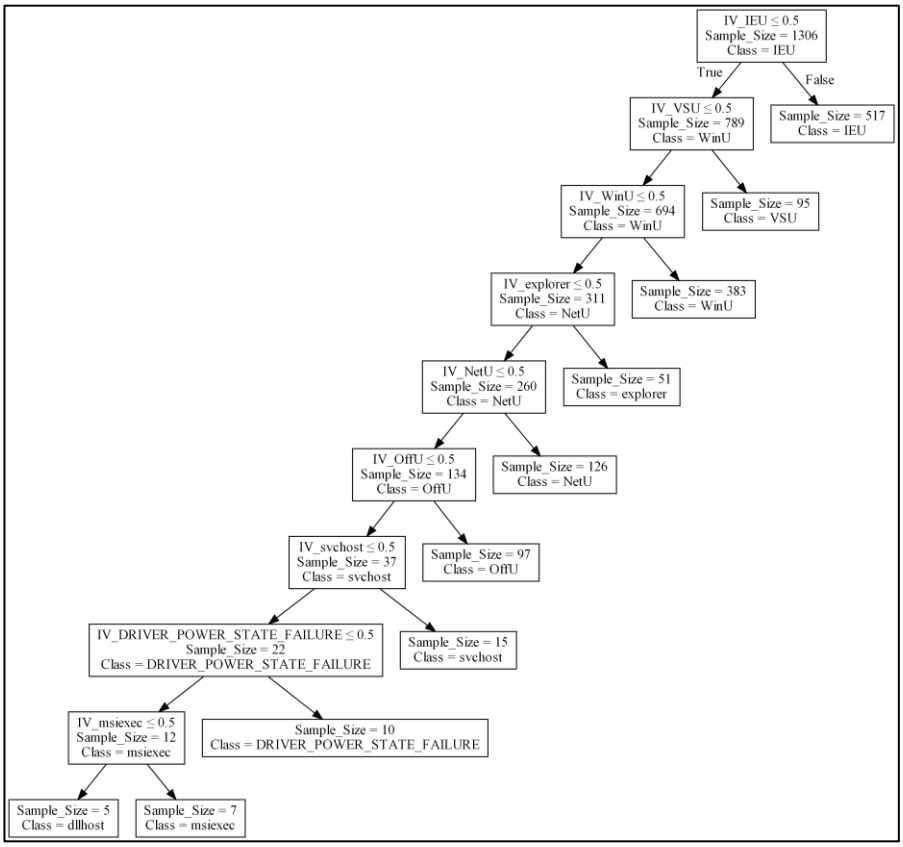


Figure A.8. Decision Tree based on sequences of two failure types (Sample_OS).

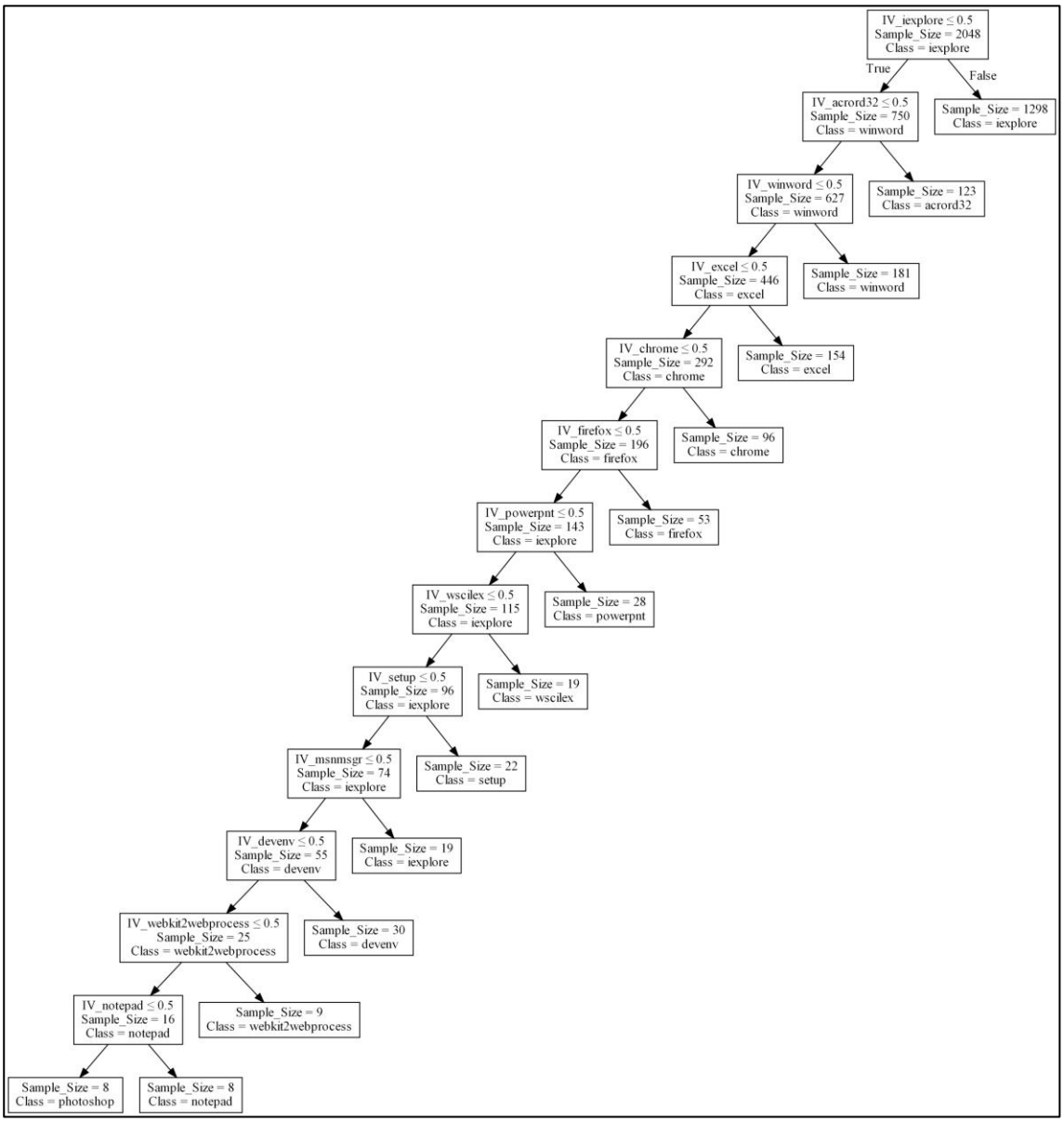


Figure A.9. Decision Tree based on sequences of two failure types (Sample_USER$_{APP}$).

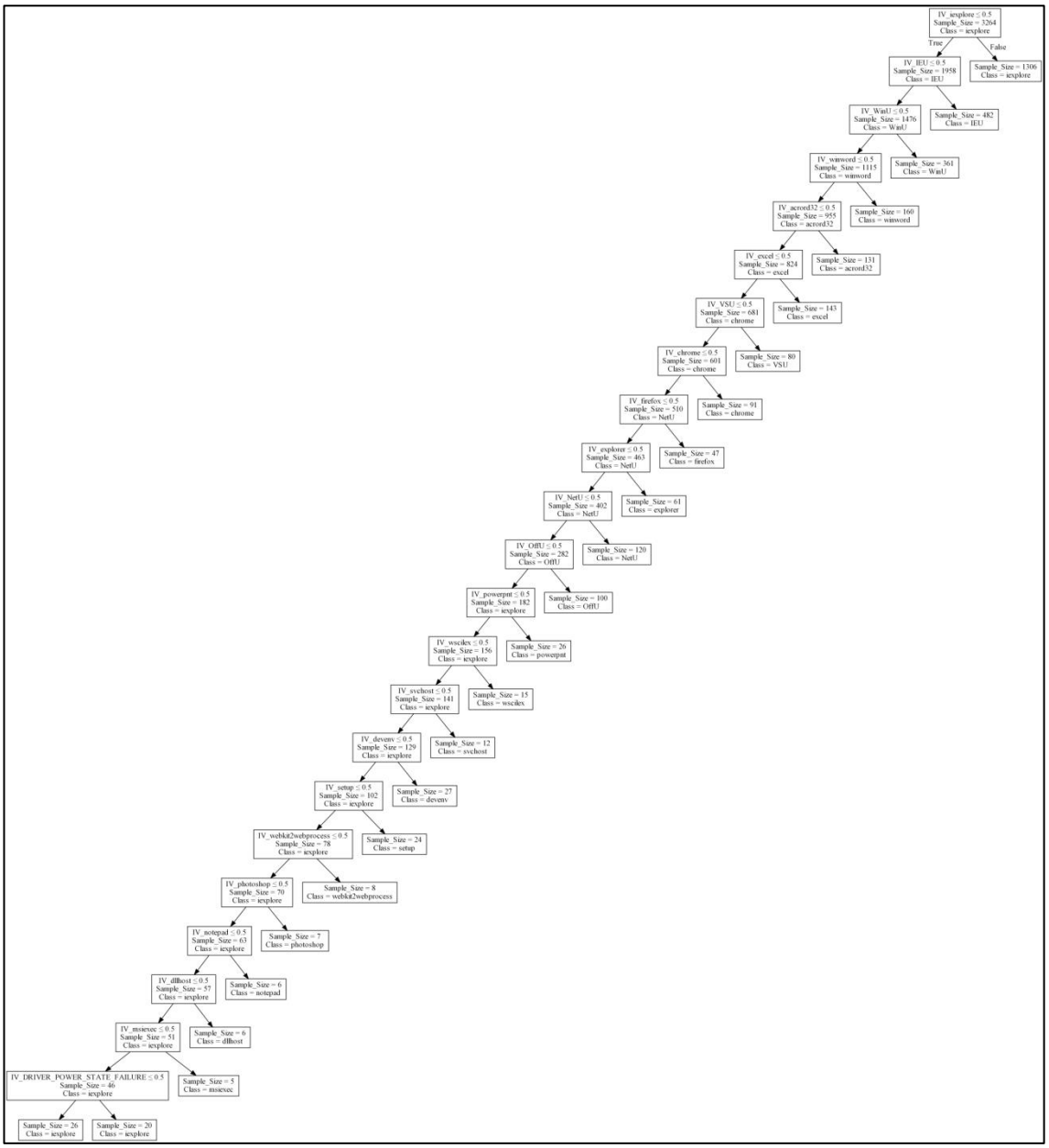


Figure A.10. Decision Tree based on sequences of two failure types (Sample_ALL).

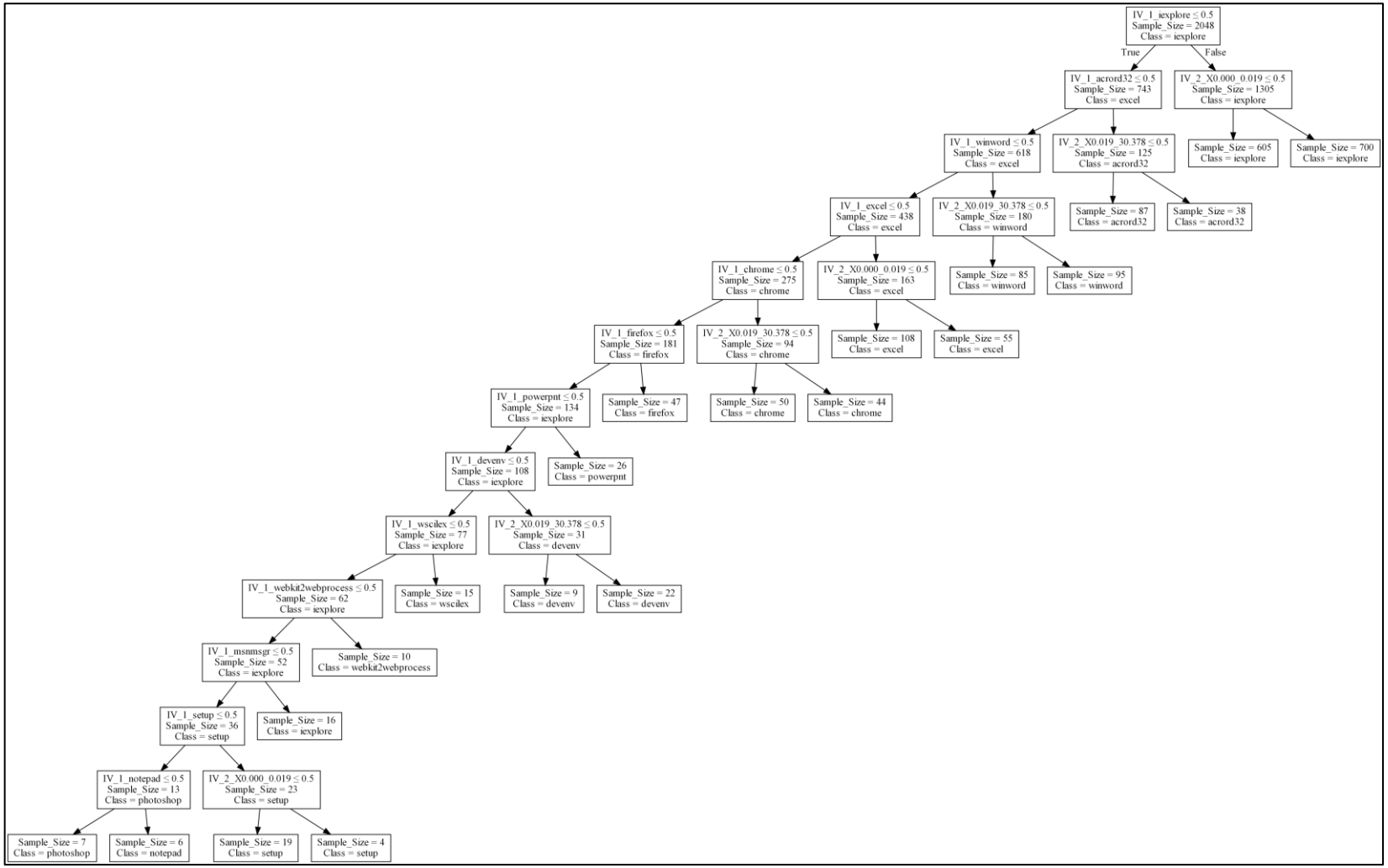


Figure A.11. Decision Tree based on sequences of two failure types and the Δt between them (Sample_USER$_{APP}$).

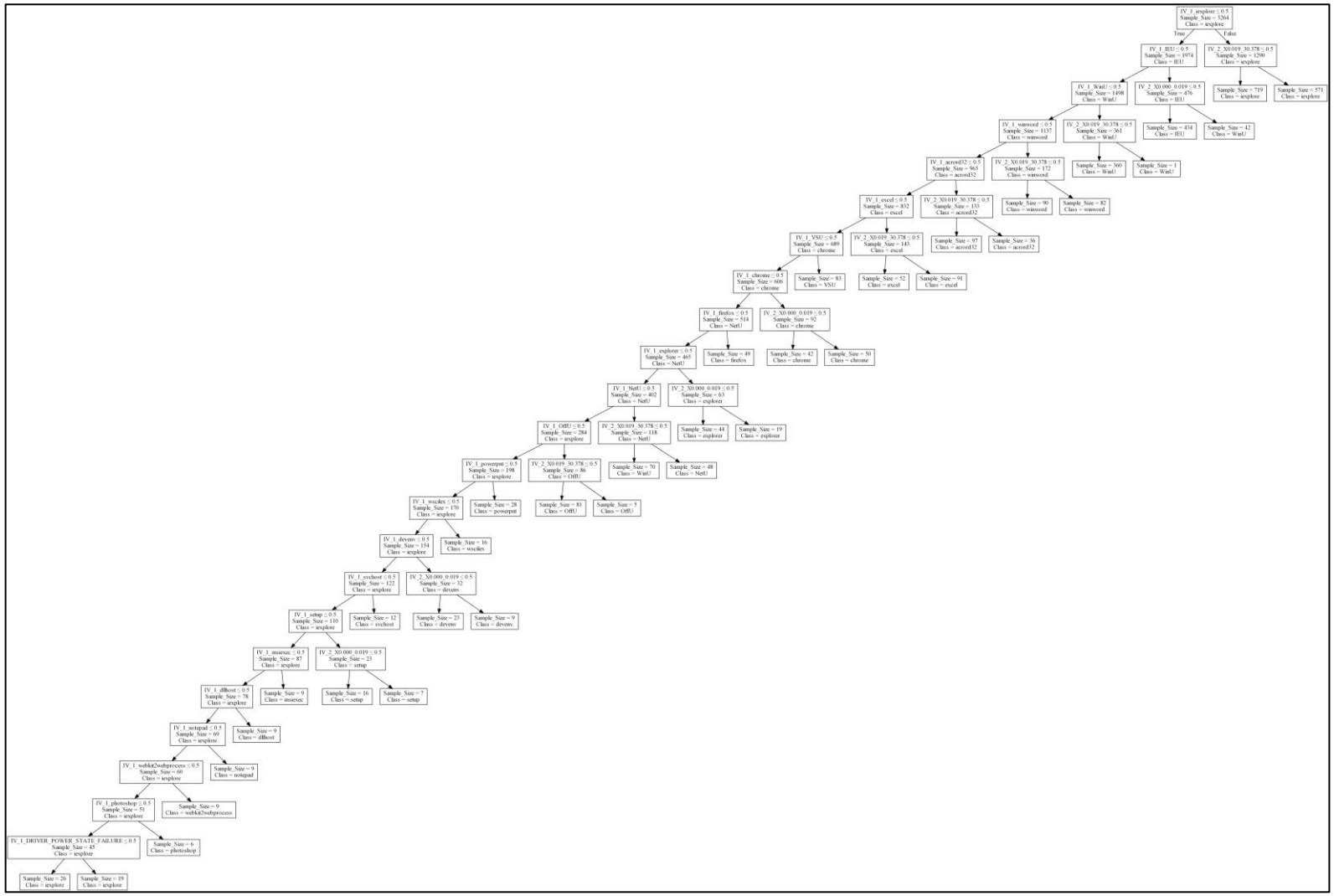


Figure A.12. Decision Tree based on sequences of two failure types and the Δt between them (Sample_ALL).

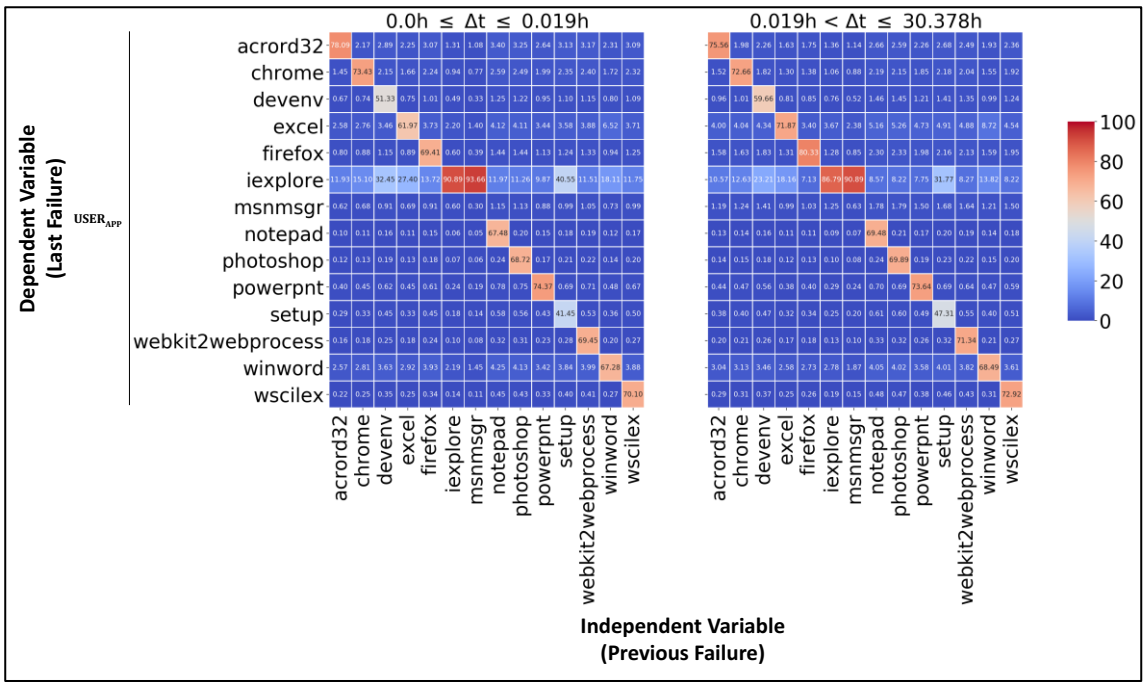


Figure A.13. Probabilities based on sequences of two failure types and the Δt between them (Multinomial Logistic Regression with Ridge - Sample_USER$_{APP}$).

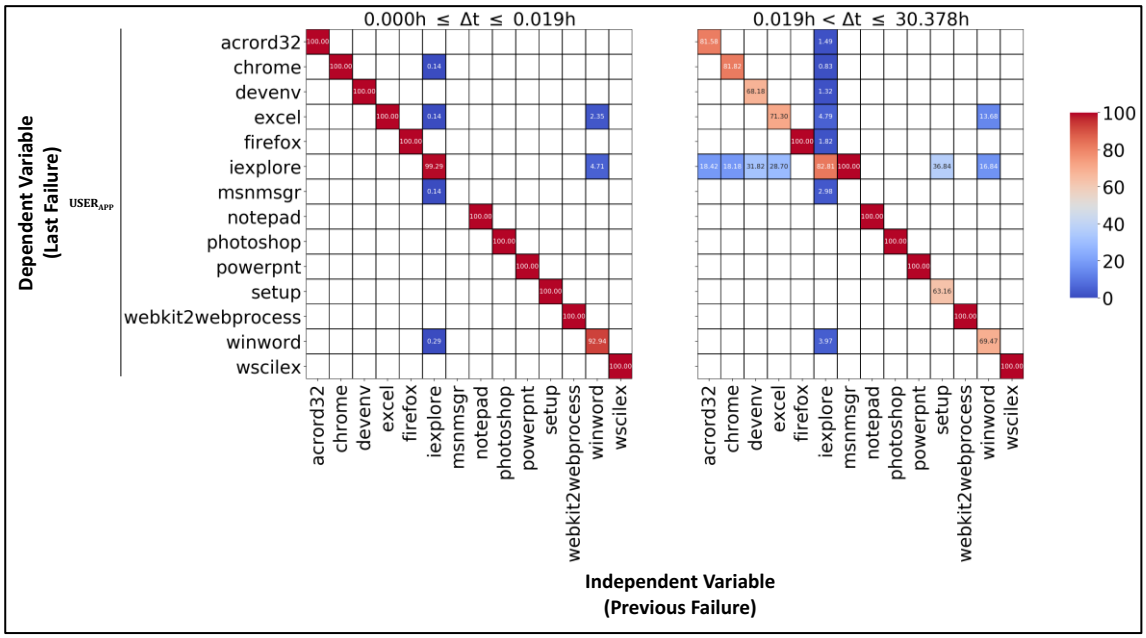


Figure A.14. Probabilities based on sequences of two failure types and the Δt between them (Decision Tree - Sample_USER$_{APP}$).

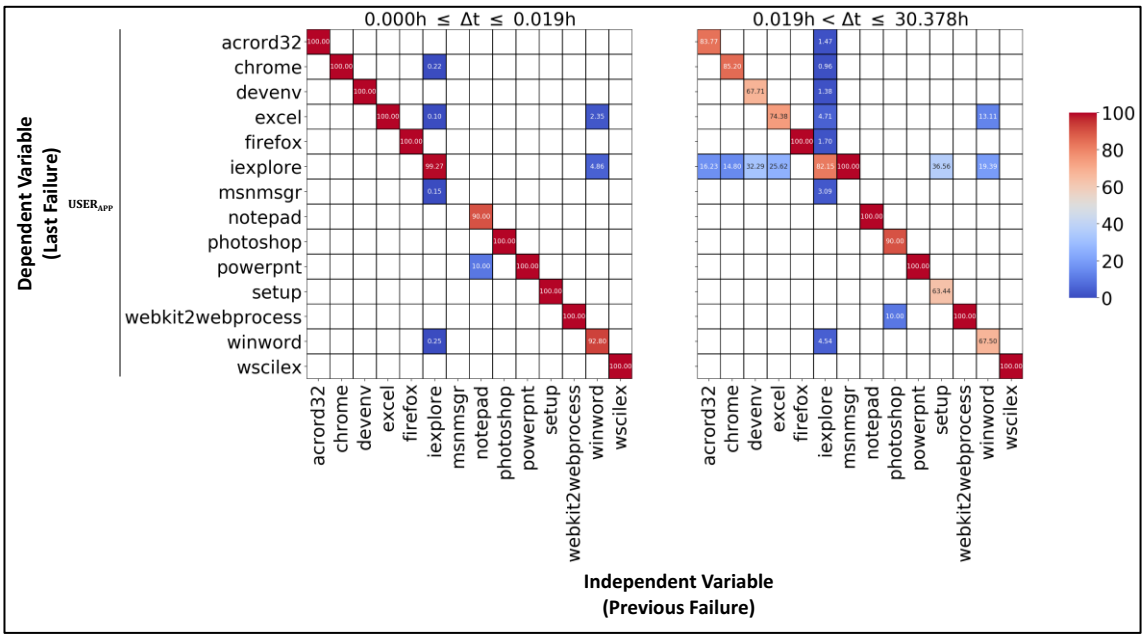


Figure A.15. Probabilities based on sequences of two failure types and the Δt between them (Random Forest - Sample_USER$_{APP}$).

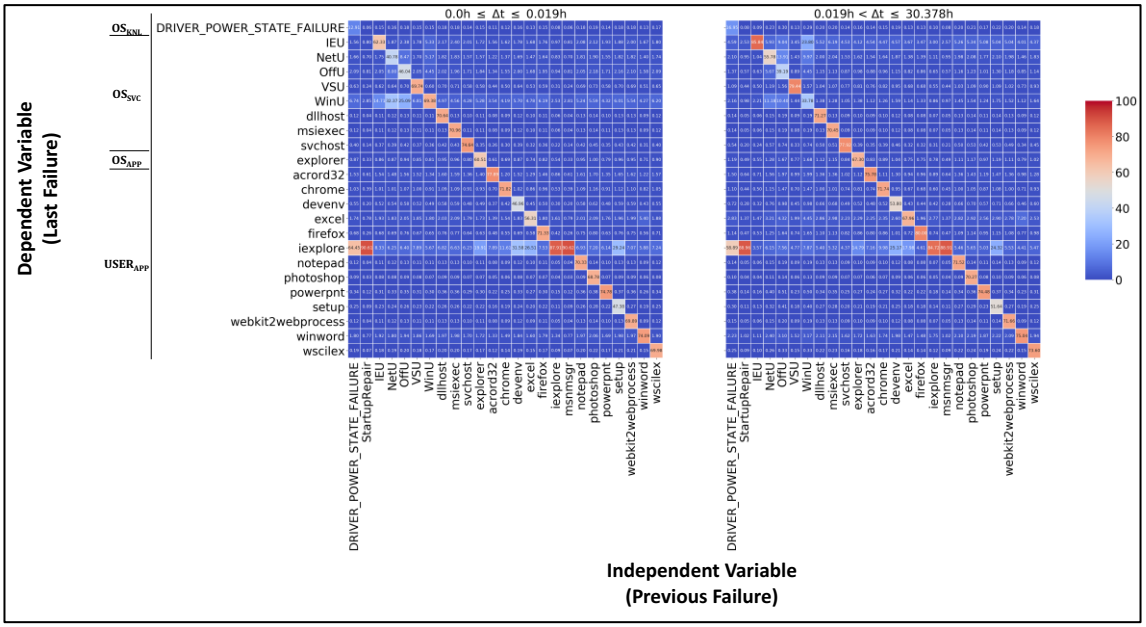


Figure A.16. Probabilities based on sequences of two failure types and the Δt between them (Multinomial Logistic Regression with Ridge - Sample_ALL).

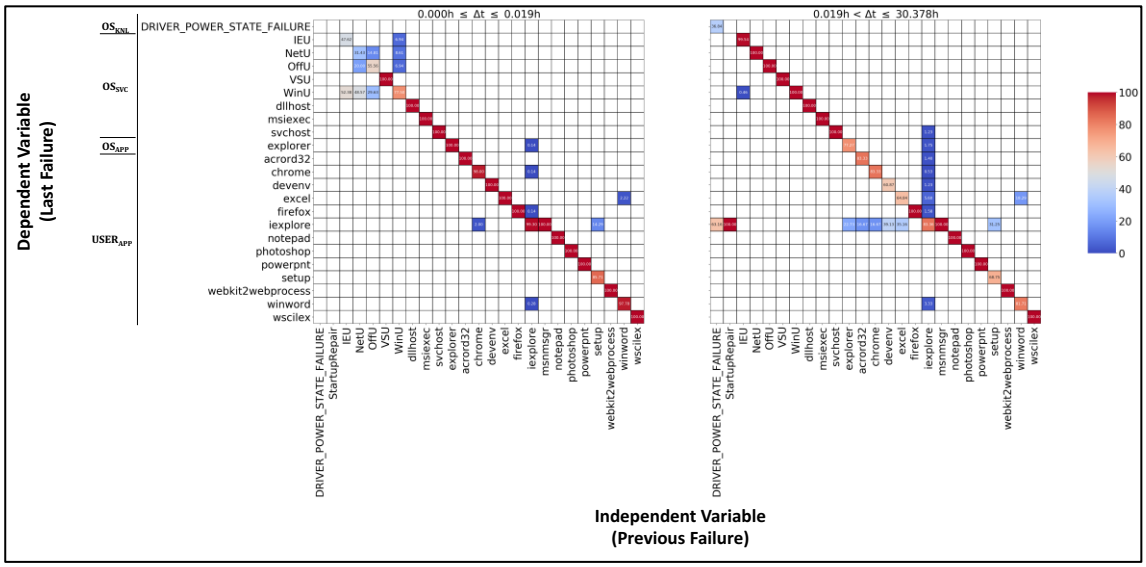


Figure A.17. Probabilities based on sequences of two failure types and the Δt between them (Decision Tree - Sample_ALL).

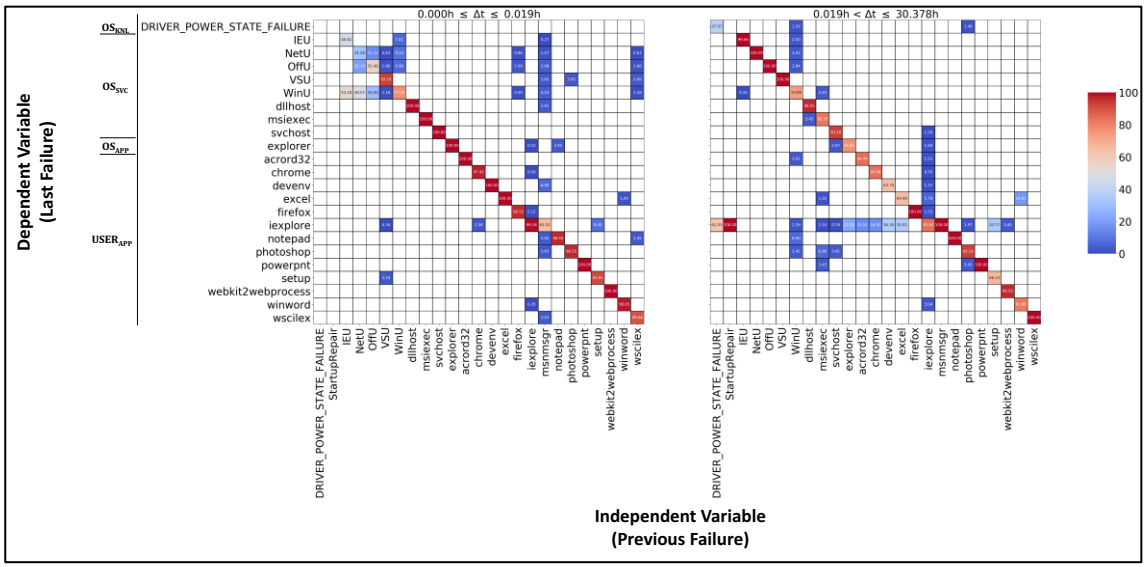


Figure A.18. Probabilities based on sequences of two failure types and the Δt between them (Random Forest - Sample_ALL).

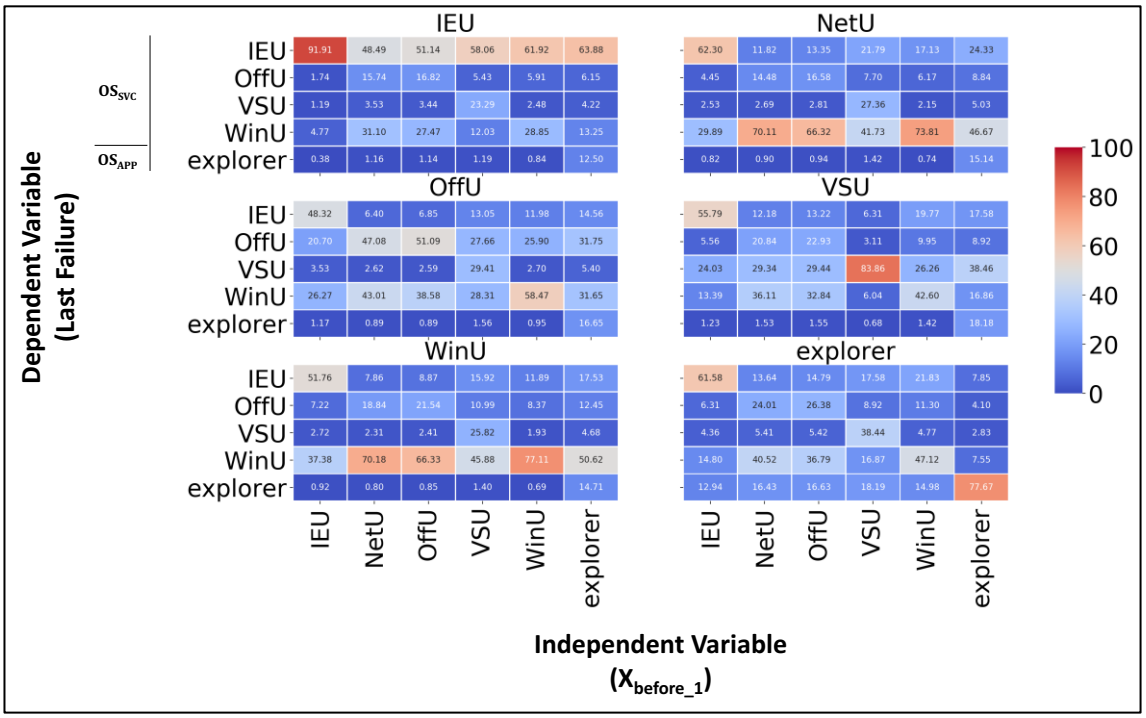


Figure A.19. Probabilities based on sequences of three failure types (Multinomial Logistic Regression with Ridge - Sample_OS).

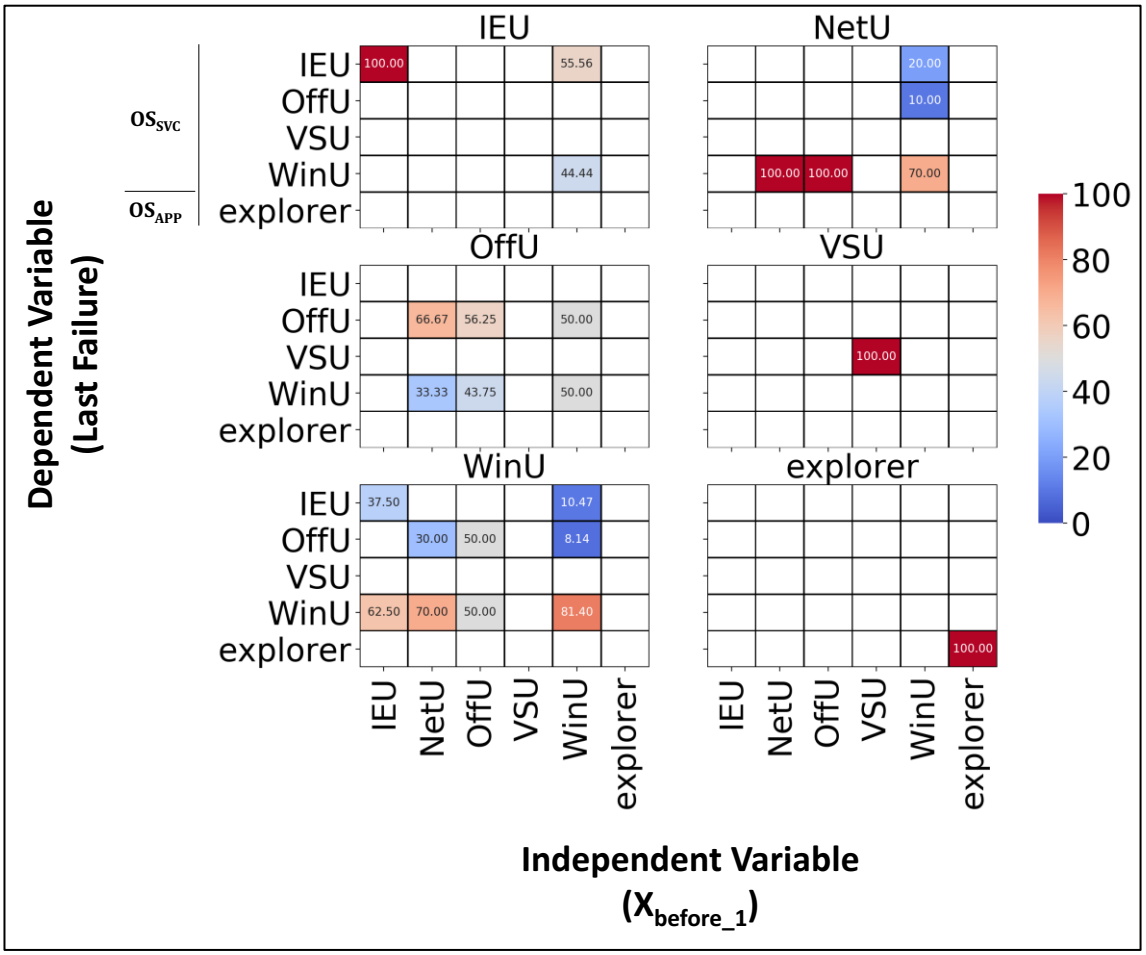


Figure A.20. Probabilities based on sequences of three failure types (Decision Tree - Sample_OS).

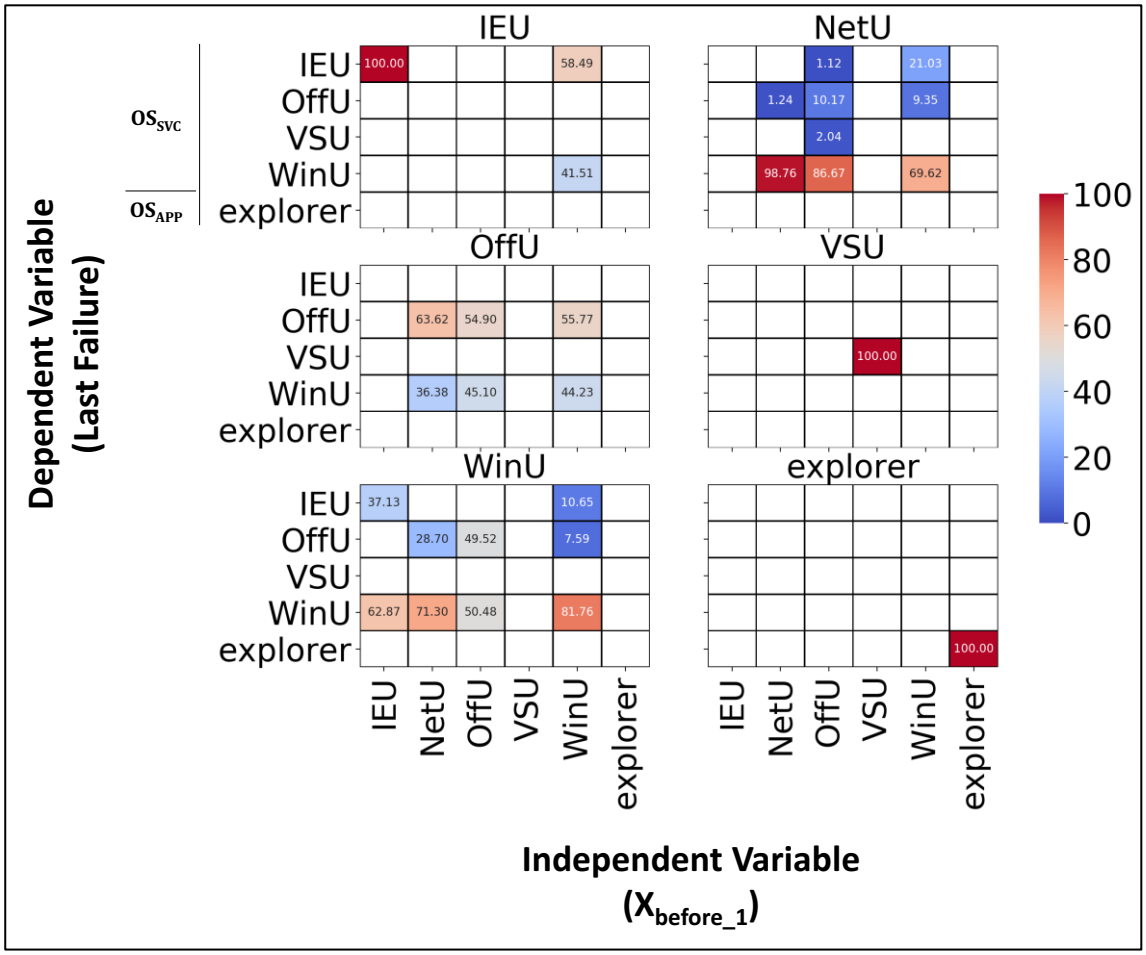


Figure A.21. Probabilities based on sequences of three failure types (Random Forest - Sample_OS).

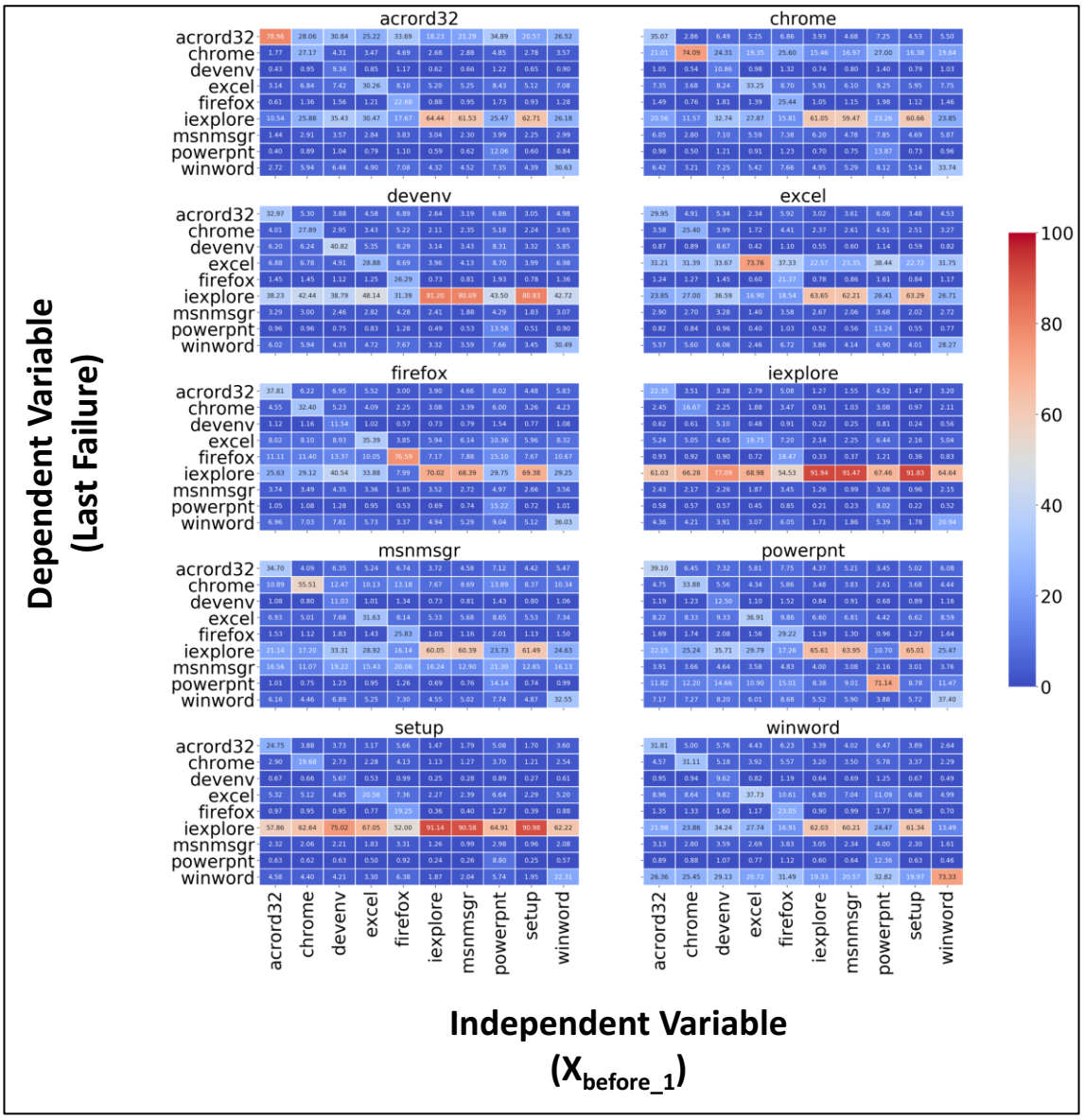


Figure A.22. Probabilities based on sequences of three failure types (Multinomial Logistic Regression with Ridge - Sample_USER$_{APP}$).

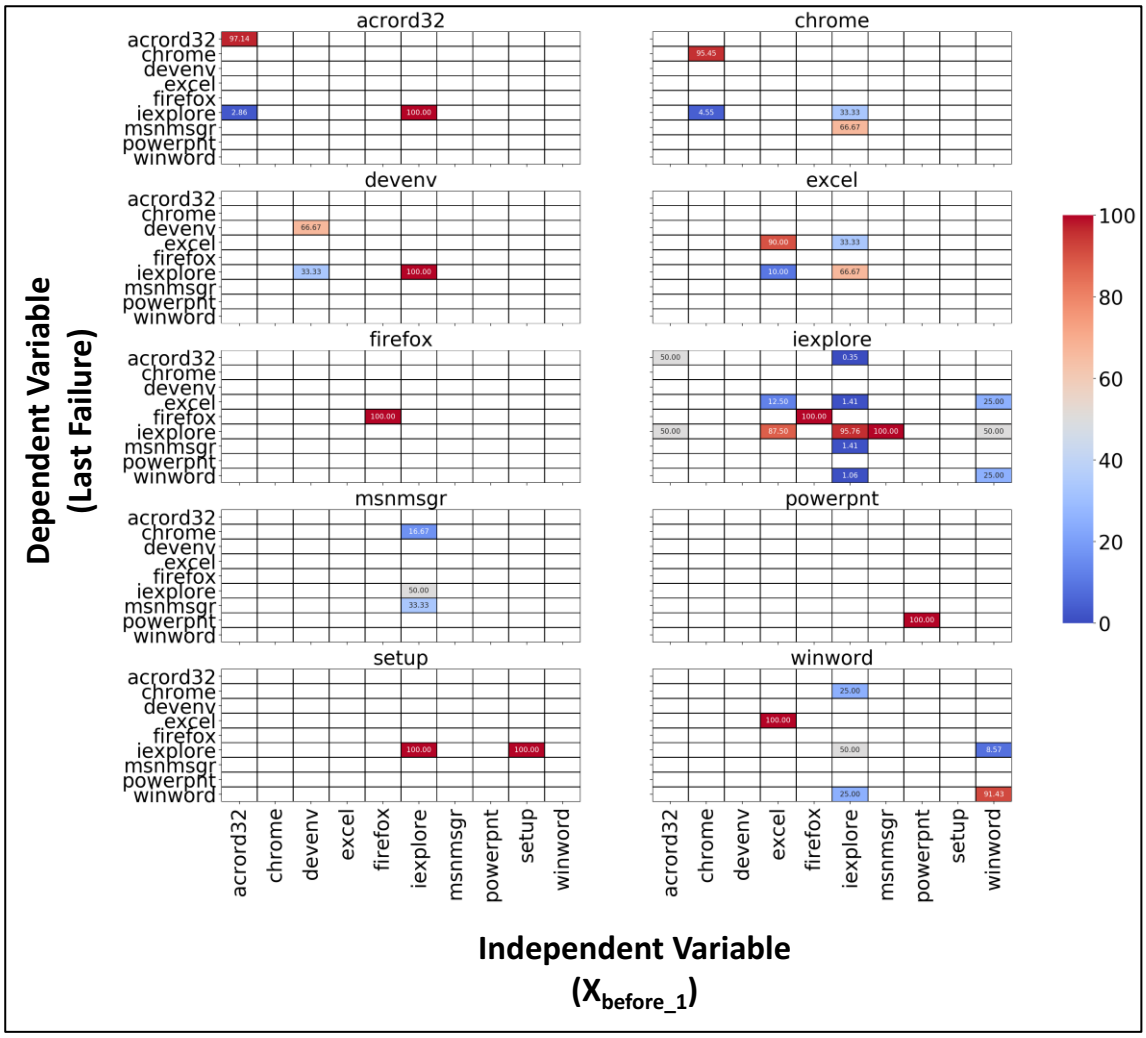


Figure A.23. Probabilities based on sequences of three failure types (Decision Tree - Sample_USER$_{APP}$).

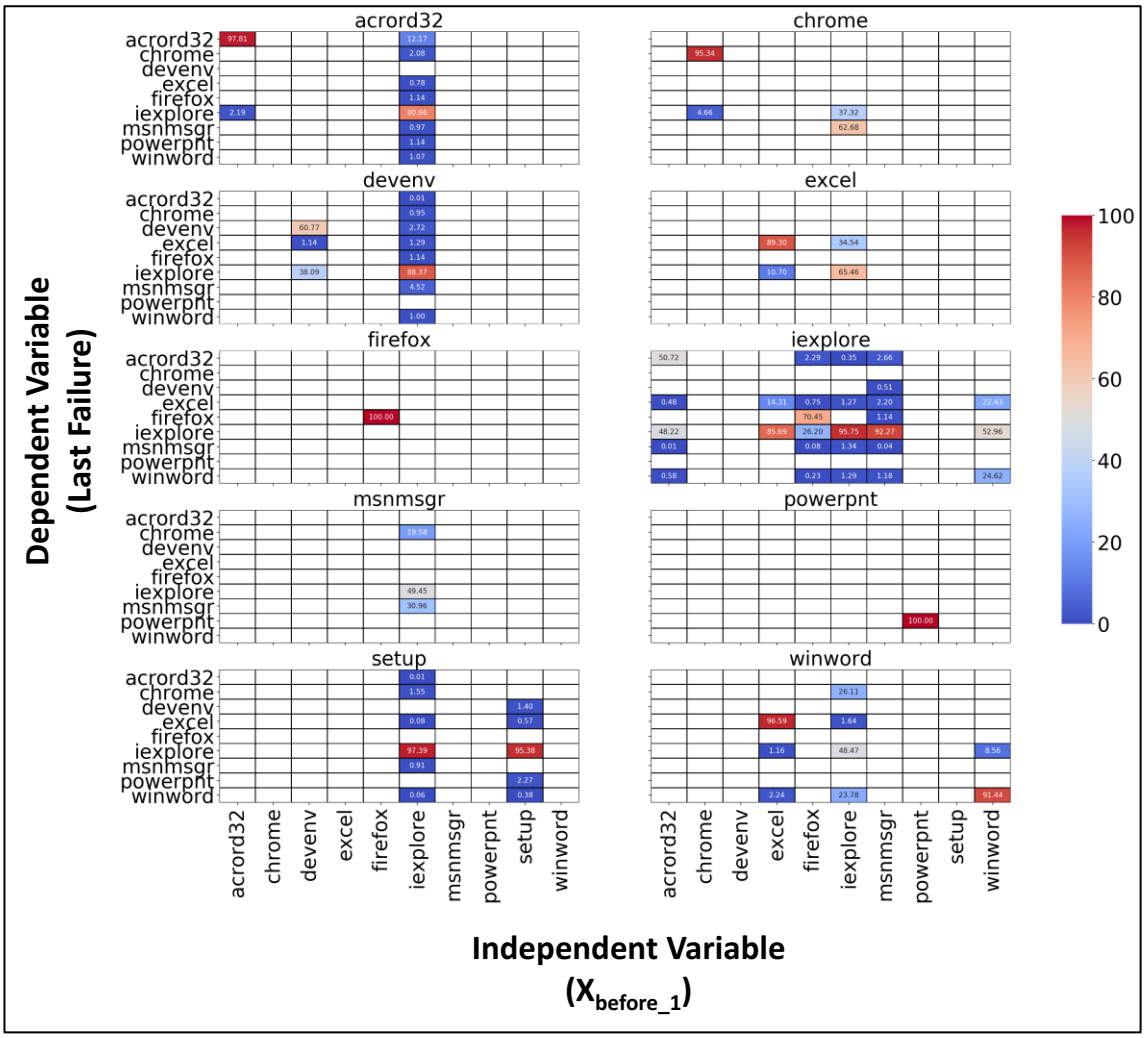


Figure A.24. Probabilities based on sequences of three failure types (Random Forest - Sample_USER$_{APP}$).

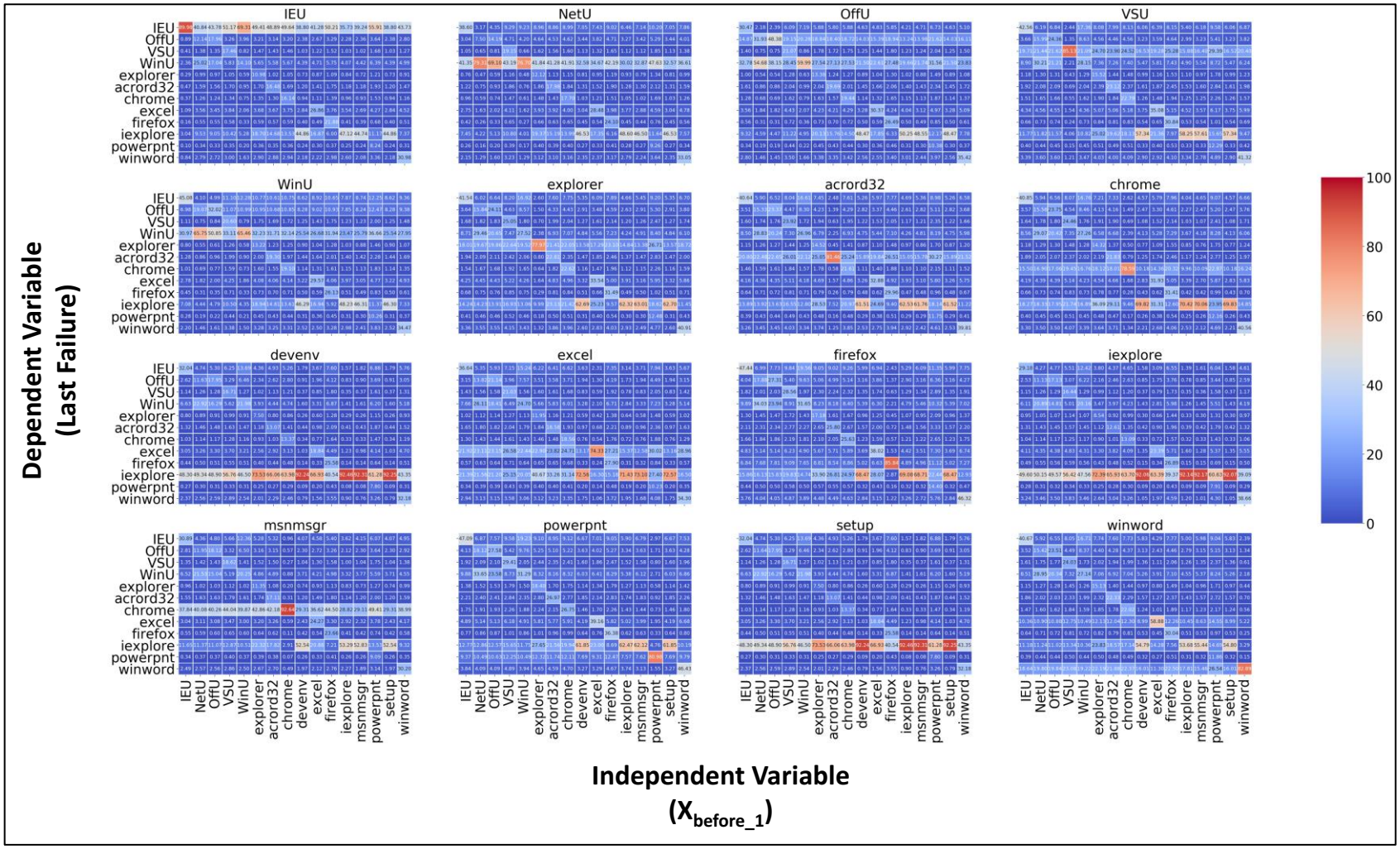


Figure A.25. Probabilities based on sequences of three failure types (Multinomial Logistic Regression with Ridge - Sample_ALL).

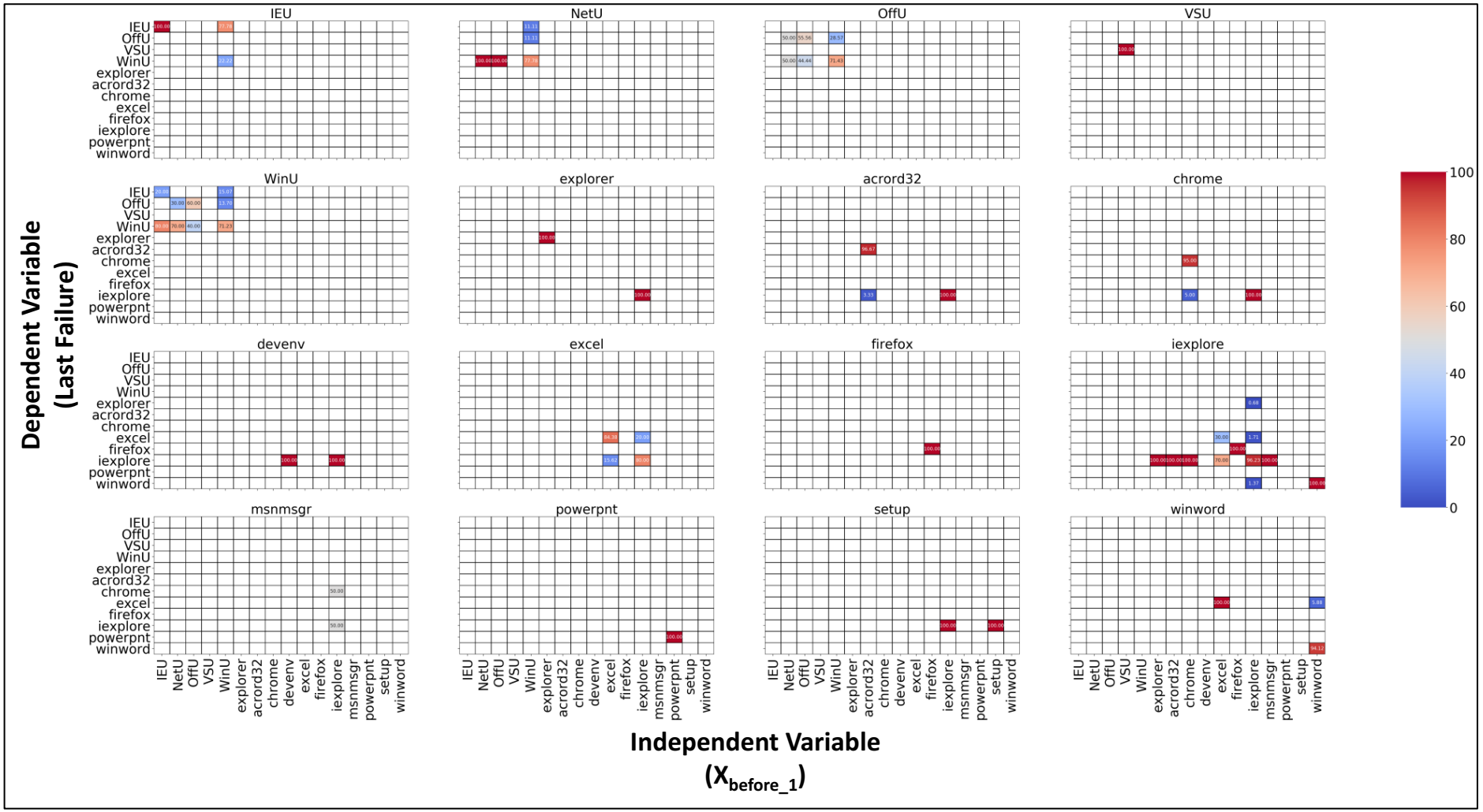


Figure A.26. Probabilities based on sequences of three failure types (Decision Tree - Sample_ALL).

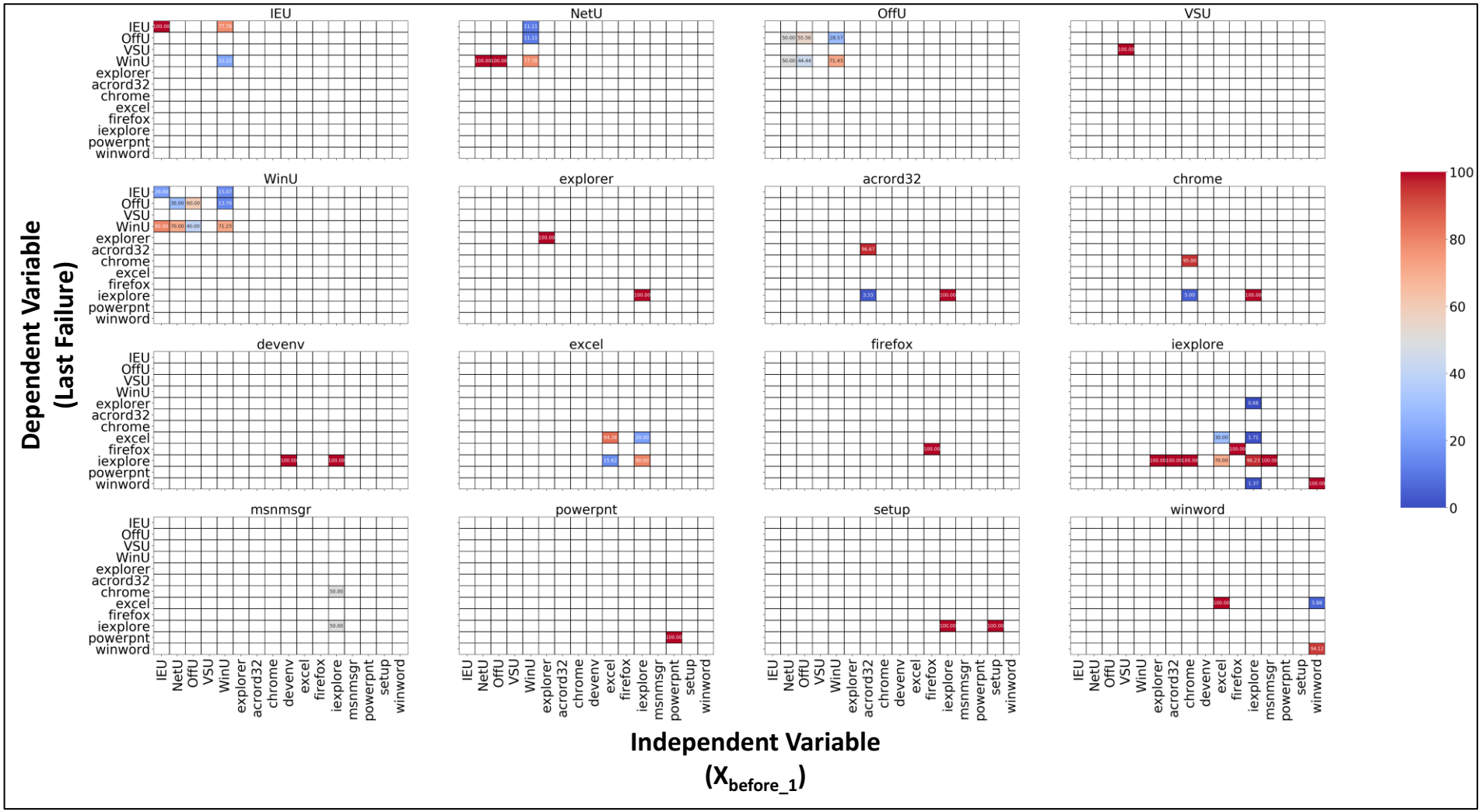


Figure A.27. Probabilities based on sequences of three failure types (Random Forest - Sample_ALL).

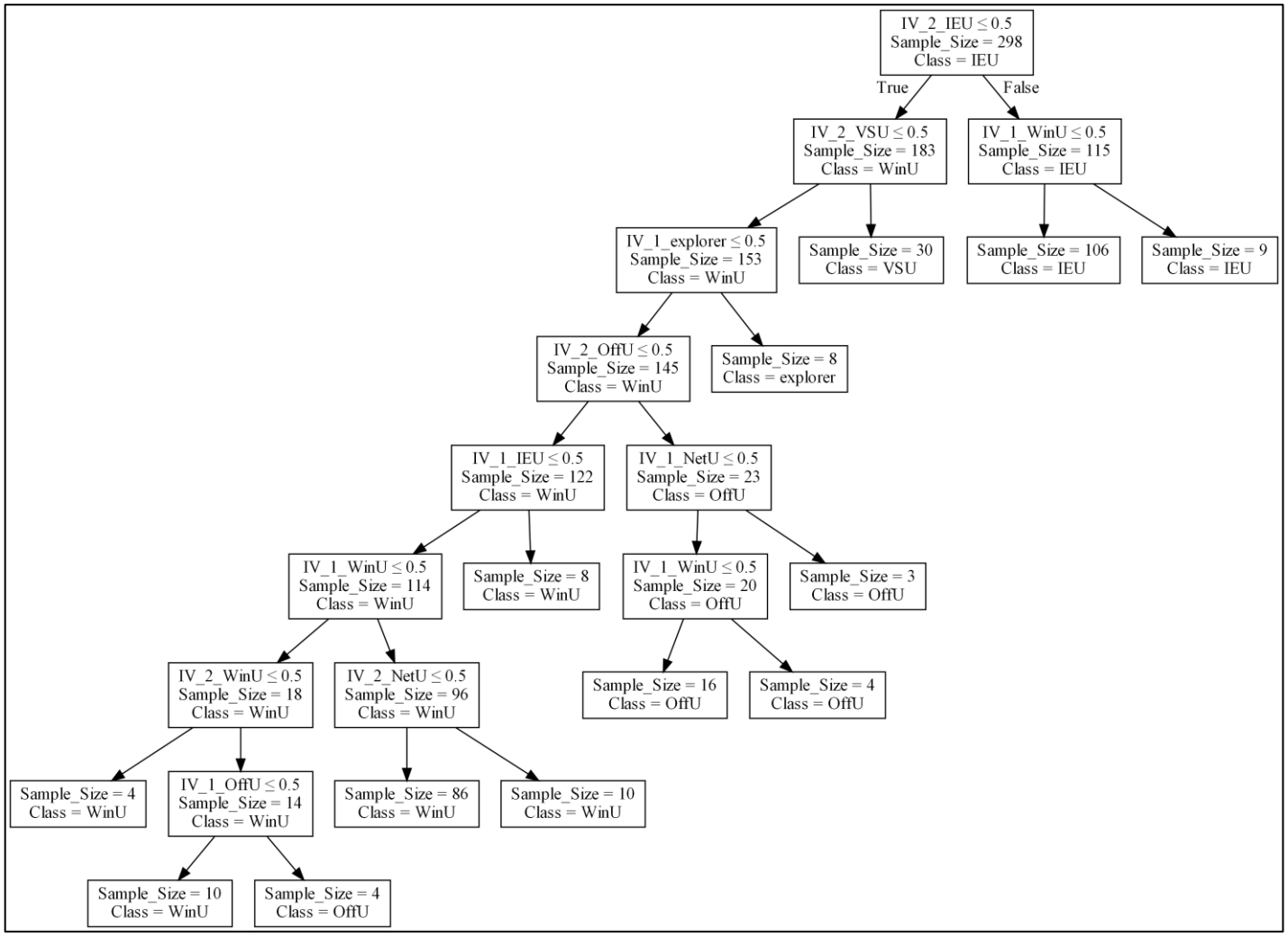


Figure A.28. Decision Tree based on sequences of three failure types (Sample_OS).

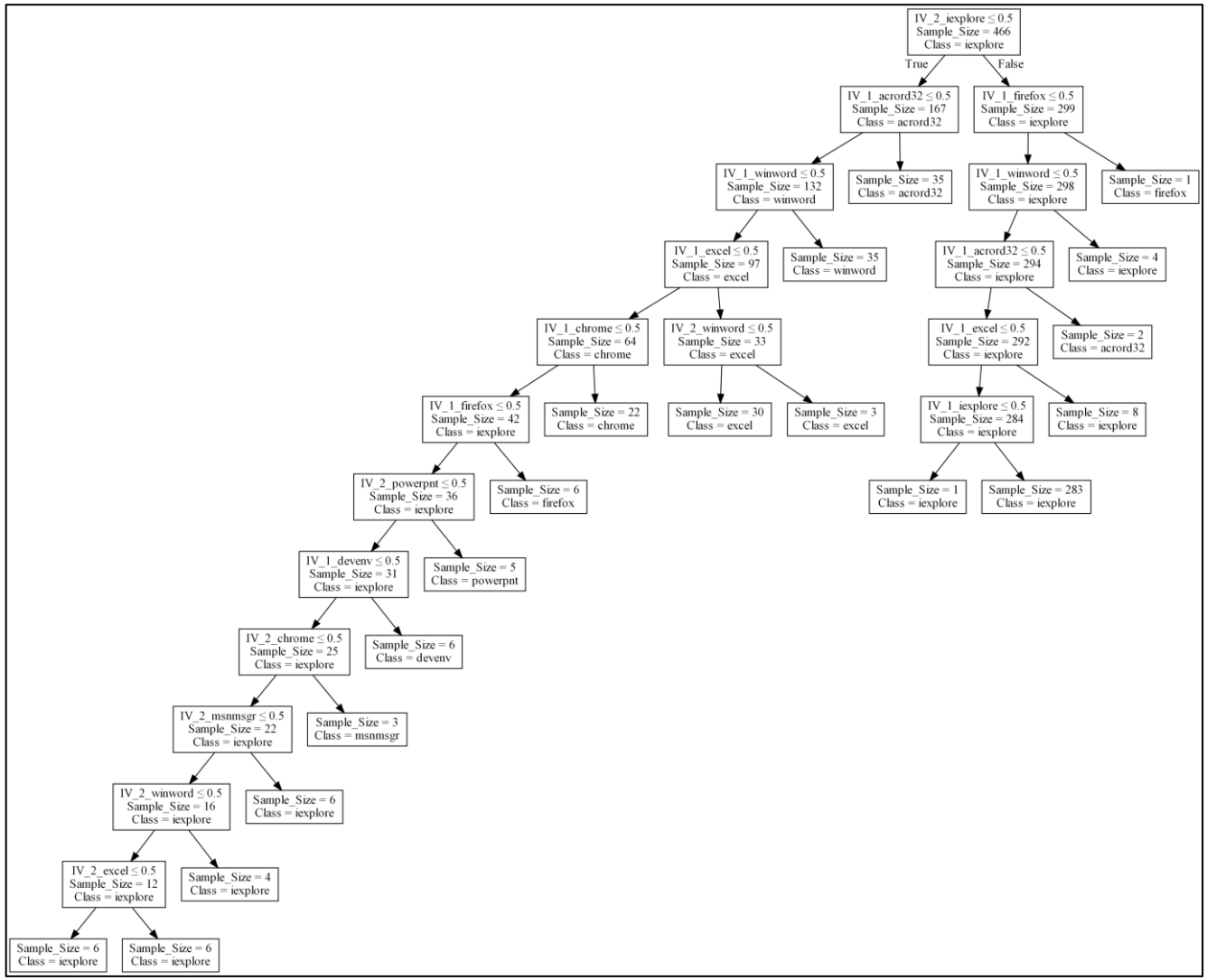


Figure A.29. Decision Tree based on sequences of three failure types (Sample_USER$_{APP}$).

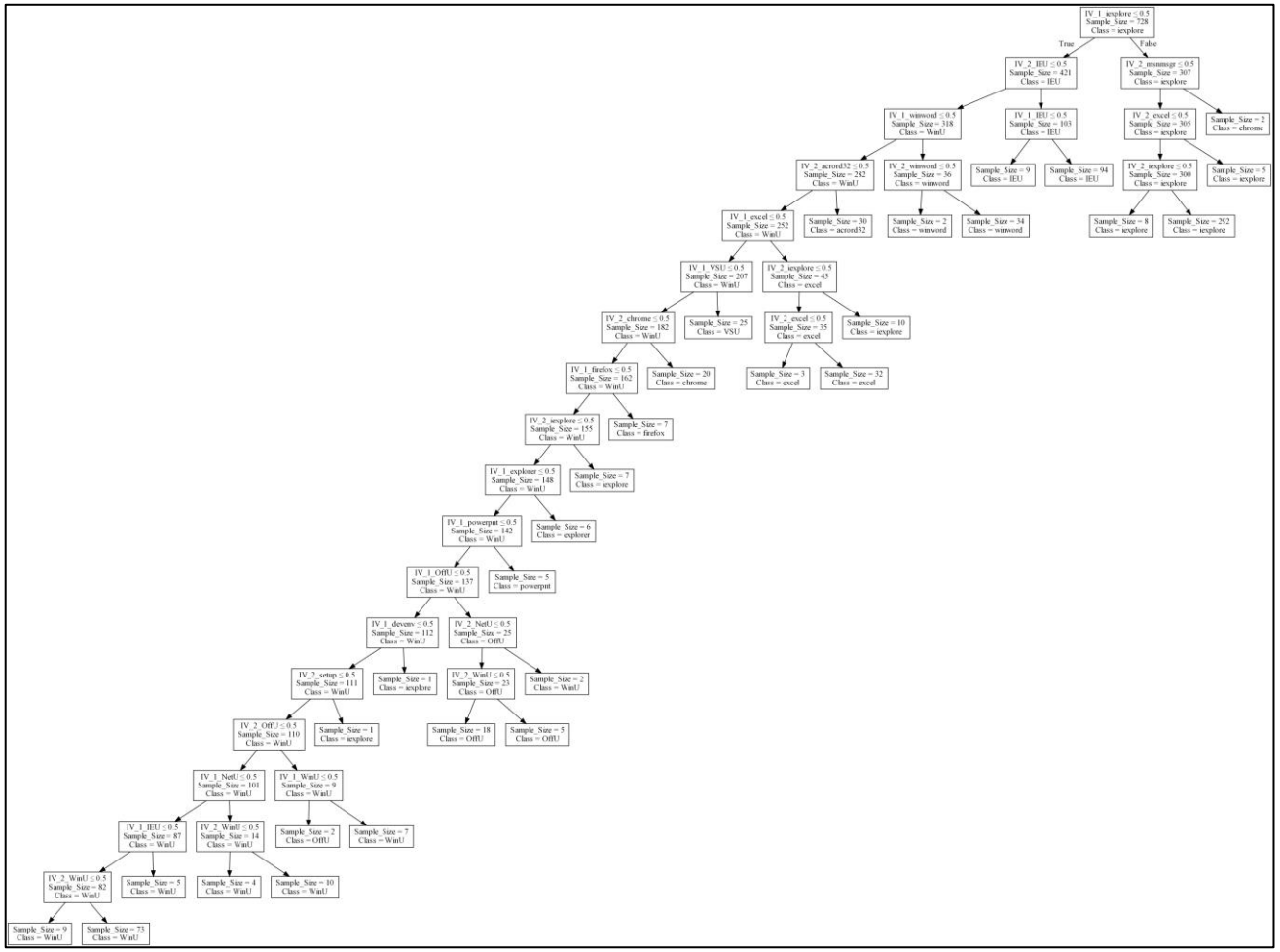


Figure A.30. Decision Tree based on sequences of three failure types (Sample_ALL).