Fault Injection for Online Failure Prediction Assessment and Improvement

Ivano Irrera

Thesis submitted to the University of Coimbra
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
August 2015

· U C ·

Department of Informatics Engineering
Faculty of Sciences and Technology
University of Coimbra

This research has been developed as part of the requirements of the Doctoral Program in Information Science and Technology of the Faculty of Sciences and Technology of the University of Coimbra. This work is within the Dependable Systems specialization domain and was carried out in the Software and Systems Engineering Group of the Center for Informatics and Systems of the University of Coimbra (CISUC).

Funding for this work was partially provided by the Portuguese Research Agency *Fundação para a Ciência e Tecnologia* (FCT) through the scholarship SFRH/BD/62538/2009.

This work has been supervised by **Professor Marco Paulo Amorim Vieira**, Assistant Professor (*Professor Auxiliar c/Agregação*), Department of Informatics Engineering, Faculty of Sciences and Technology, University of Coimbra.

"Essentially, all models are wrong,
but some are useful"
George E. P. Box

"Empirical Model Building and Response Surfaces",
Box, G. E. P., and Draper, N. R., (1987),
John Wiley & Sons, New York, NY,
p. 424

To my family,
to my friends

Abstract

Software is the fundamental brick of the systems that pervade our society, such as communications, transportation, business and health caring. In fact, software is a mean for building and controlling increasingly complex business processes, with an unlimited potential. However, complex software hides defects (i.e., *software faults*) that may lead to the occurrence of failures affecting the business. Several techniques allow decreasing the number of software faults (e.g., testing) or reacting to the occurrence of failures (e.g., fault tolerance). Nonetheless, it is well known that failures are ultimately unavoidable events that may cause loss of data, performance, or even money or human lives.

A solution for limiting the damage caused by system failures is to predict their occurrence by analyzing the system and observing its state. Failure Prediction is a technique proposed in the past to predict failures by analyzing the system architecture and the development processes, or by learning from past failure data (e.g., the time between successive failures). Such technique evolved into **Online Failure Prediction**, which correlates past failure data with the current system state, increasing the quality of the prediction. In practice, the prediction of an incoming failure allows performing mitigation actions, such as saving data or restarting parts of a system, to lessen possible hazards.

Despite its potential, Online Failure Prediction is still not widely adopted. The main reason is that failures are rare events and the collection of the failure data needed for training a predictor is a non-controllable process that takes a long time and has a high cost. This becomes even more evident if we consider that current software systems are dynamic in nature and that the failure data collected today may not portray the behavior of the system tomorrow. In fact, the difficulty in collecting failure data is the main reason why Online Failure Prediction has not been used in practice so far, as training, optimizing and validating failure prediction models becomes very hard to achieve.

This work addresses the current limitations of Online Failure Prediction by taking advantage of fault injection techniques to speed up the occurrence of failures. The thesis is that software fault injection is a valid solution to generate failure-related data in short time for a particular system, promoting the use of Online Failure Prediction by helping in training, optimizing and validating different prediction models.

First, we study the conditions under which software fault injection can be used to support failure prediction and proposes an approach to generate failure-related data and assess their accuracy. Then, we propose a method for assessing and comparing different failure prediction models in the context of a particular target system, and present a framework for self-adapting online failure prediction systems based on the continuous generation of failure data on a virtualized copy of the target system, thus facing the dynamic features of current software systems. A preliminary method for selecting the best variables for predicting failures is also presented.

To validate and demonstrate the different techniques and tools, we present a number of case studies. The target system used in the case studies is based on a Windows XP OS running several different workloads, ranging from a simple file compression algorithm, up to a web server. This diversity allows collecting insights on the impact of the workload on the failure data generation and failure prediction. The web server workload, made of the widely used Apache Tomcat web server, provides a realistic scenario, which is used for analyzing the impact of updates on the prediction of failures on the system. The reason that stays behind the choice of Windows XP is that it was a widely spread and stable operating system whose failures are well known, thus being a good environment for analyzing failures caused by injected faults. Results clearly show that fault injection can be used to improve the state-of-the-art on failure prediction.

Keywords

Online failure prediction, software fault injection, benchmarking, dynamic systems, virtualization.

Resumo

A cada vez maior complexidade do software faz com que muitos sistemas sejam usados contendo defeitos (i.e., falhas de software) que podem levar à quebra do seu correto funcionamento (i.e., avarias). Várias técnicas permitem a mitigação dos efeitos das falhas de software, diminuindo o número de falhas (por exemplo, através de testes), ou reagindo à ativação das falhas existentes (por exemplo, através técnicas de tolerância a falhas). No entanto, as avarias são eventos inevitáveis na vida de um sistema complexo e podem levar à perda de dados, desempenho, dinheiro ou até vidas.

Uma solução para mitigar os danos causados por falhas de software consiste em prever a ocorrência de avarias através da análise do sistema, em particular a observação do seu estado interno. Esta técnica, denominada de previsão de avarias (*Failure Prediction*), foi inicialmente baseada na análise da arquitetura e do processo de desenvolvimento do sistema, podendo também considerar dados históricos sobre o seu funcionamento (por exemplo, o tempo de intercorrência entre duas falhas sucessivas). Mais recentemente, evoluiu-se para a previsão de avarias em tempo de execução (**Online Failure Prediction**), em que os dados sobre o funcionamento passado de um sistema são correlacionados com o seu estado atual, de forma a obter uma melhor qualidade na previsão. Na prática, prever a ocorrência de uma avaria permite executar ações de mitigação e diminuir os riscos, como por exemplo gravar os dados ou reiniciar partes do sistema.

Apesar do seu potencial, a previsão de avarias em tempo de execução é ainda pouco utilizada. A principal razão reside na dificuldade em treinar os mecanismos de previsão, já que tal requer a recolha de dados relacionados com avarias observadas no passado. Este processo requer tipicamente demasiado tempo e não é controlável, resultando num custo elevado. Isto torna-se ainda mais evidente se considerarmos que os sistemas de software atuais são dinâmicos por natureza: isto é, evoluem ao longo do tempo levando a que os dados recolhidos num intervalo de tempo se tornem rapidamente obsoletos. De facto, a dificuldade em recolher dados é a razão principal para que as técnicas de previsão de avarias em tempo de execução não sejam ainda utilizadas na prática, uma vez que o treino, otimização e validação dos modelos de previsão se tornam difíceis de realizar.

Este trabalho aborda as limitações atuais da previsão de avarias, através da utilização de técnicas de injeção de falhas de software para acelerar a ocorrência de avarias. A hipótese é que a injeção de falhas é uma solução válida para gerar dados de avarias em um sistema durante um curto intervalo de tempo, suportando assim o treino, otimização e validação de diferentes modelos de previsão.

Em primeiro lugar, são estudadas as condições nas quais a injeção de falhas de software pode ser utilizada para suportar a previsão de avarias, sendo proposta uma abordagem para a geração de dados e para a avaliação da representatividade desses mesmos dados. De seguida, é proposto um método para avaliar e comparar diferentes modelos de previsão no contexto de um sistema específico e apresentada uma solução para a auto-adaptação de sistemas de previsão de avarias capaz de acompanhar a evolução do sistema. A solução proposta assenta na geração contínua de dados em uma cópia virtualizada do sistema. Para além disso, é proposto um método para selecionar as melhores variáveis para suportar o processo de previsão.

Para demonstrar e validar as diferentes técnicas e soluções propostas, são apresentados vários casos de estudo. Os sistemas utilizados são baseados no sistema operativo Windows XP e incluem a execução de diferentes cargas de trabalho, desde um simples algoritmo de compressão de arquivos, até um servidor Web. Esta diversidade permite estudar o impacto da carga de trabalho na geração de dados. Os resultados mostram a aplicabilidade da injeção de falhas de software no contexto da previsão de avarias em sistemas de computadores.

Palavras-chave:
Previsão de avarias, injeção de falhas de software, testes padronizados, sistemas de software dinâmicos, virtualização.

Acknowledgements

To instill words with all the gratitude for the people and friends that walked at my side during the last years is the hardest part of this work. I am sure the following words are bound to fail, which however is good, as it gives me the possibility to dedicate to you my past and future steps, with all my gratitude and love.

I would like to start by thanking to Professor Marco Vieira for his guidance through the entire path that led me to this point, for his time and patience, for his support in the difficult moments, and for the inspiration that he is for my professional life. Most important, I thank him for his friendship.

I thank the members of the Software and System Engineering group of the Centre of Informatics and Systems of the University of Coimbra for the excellent quality of their work, an everyday inspiration. I would also thank my colleagues of the SIGDep group, for their constant strive for excellence and the inspiring scientific discussions.

I thank all the anonymous reviewers that helped me to improve this work with their comments.

I would like to thank all my friends and colleagues for simply being at my side and for supporting me. Without them I wouldn't have achieved my professional goals. Most important, I would not be the person that I am.

To Nuno, to which I owe so much as a friend and a colleague, a continuous force and inspiration, with whom I am proud to have shared this great adventure here in Portugal, hoping to share more adventures in the future.

To Ivano ("2"), a piece of home at my side, a reference in the difficult days and a friend with whom I shared many unforgettable moments, and to whom I owe more than this.

To João, a gentle light in the dark, always ready for helping and supporting me as a professional and as a friend, to whom I owe many of my achievements.

To Rafa, Pedro ("Correia"), Nito, Cristiana, Marco, for all the unforgettable moments. Let more come in the future! I must also thank the help, the good humor

and the distractions of my laboratory colleagues: to Rui, Diogo, Ana, Luis, Ivo, and Sergio, for the many good moments.

To all my friends Andréa, Paulo G., Eudis, Luciana, Suhely, Amanda, Marcelo, Gabi, Gustavo, Renata, Isabella, Karina, Tamara, for their friendship, the unforgettable moments lived together, and their help in many hard moments. Never before I thought to know so many brasilian in my life never before I thought that the world could be so surprising. To Heitor, Bruno, Hudson, and the people with whom I shared my time at home here in Coimbra. To them I am thankful for the richness that they brought to my life.

To my friends Naaliel, David, Davi, Sven ("o alemão"), Nuno L., Paulo V., Amir, Merk, with whom I shared my path here in Coimbra. To Naghmeh, friend since the beginning of my Ph.D. adventure, to whom I owe this moment: without those words, back in 2012, I would probably be in a different place at this time.

To all the people I met during these years in Coimbra and around the world, hoping to see them again soon. I would like to thank Christian, for his advices, support and friendship, and for the funny moments lived here in Portugal. Special greetings go to Fumio, Andreas and Andrea C., for their worldwide good humor and friendship.

To all the guys of the DEI with which I shared many coffees, “Ph.D. dinners” and fast encounters, full of laughs and funny moments, goes my deepest gratitude.

I would like to express my gratitude to my family, for their love. I thank my parents Nunzio e Adele for their sacrifices and continuous hard work that gave me the possibility to achieve this goal, and to see with me the wonders of the world we live on. I thank for their unconditional love in the good and bad moments, and for all the moments that we lived. I owe you my life.

To Simona and Dimitri, my sister and brother, to whom I owe more than I can express, I thank for the always good and funny moments, full of life and joy. I cannot imagine how my life would be without them.

To my cousins Moreno, Danilo and Francesca, for the many good moments, the infinite support, and the love that make us more than cousins.

To all my friends back at home, for their uninterrupted love and support, I owe my deepest part, hoping to continue sharing many moments in the future.

I thank Coimbra, for the many moments I passed here and the lessons learned during these years. I thank Portugal for being my second home, for the kindness and welcoming of its people, and for the moments that were and will be.

I thank everyone for procrastinating my work, with a special thanks to “Porta dos Fundos” group: without that I cannot imagine what would be of us. I thank artists for their constant work for keeping us human. May always distractions and procrastination lead and save our lives.
Agora e sempre, carrega Benfica!

Ringraziamenti

Infondere queste parole di tutta la gratitudine per le persone e gli amici che hanno camminato al mio fianco durante questi ultimi anni è la parte più difficile di questo lavoro. Sono sicuro che le parole che seguono sono destinate a fallire, il che tuttavia è una cosa buona, perché mi dà la possibilità di dedicarvi i miei passi, passati e futuri, con tutta la mia gratitudine e amore.

Vorrei innanzitutto iniziare col ringraziare il professor Marco Vieira per la sua guida e orientamento lungo l'intero percorso che mi ha portato fin qui, per la sua pazienza e il suo sostegno nei momenti difficili, e per l'ispirazione che è per la mia vita professionale. Cosa più importante, lo ringrazio per la sua amicizia.

Ringrazio i membri del gruppo di SSE del CISUC dell'Università di Coimbra per l'eccellente qualità del loro lavoro, una fonte d'ispirazione quotidiana. Vorrei inoltre ringraziare i colleghi del gruppo SIGDep, per la loro costante ricerca dell'eccellenza e le costruttive discussioni: una costante fonte d'ispirazione. Ringrazio tutti i revisori anonimi che con i loro commenti hanno aiutato a migliorare questo lavoro.

Vorrei ringraziare tutti i miei amici e colleghi semplicemente per essere al mio fianco e per avermi supportato durante tutto questo tempo. Senza di loro non solo non avrei raggiunto i miei obiettivi professionali, ma non sarei la persona che sono.

A Nuno, al quale devo tanto come un amico e come collega, per la sua forza e ispirazione continua, con il quale sono orgoglioso di aver condiviso questa grande avventura qui in Portogallo, nella speranza di condividere più avventure in futuro.

A Ivano ("2"), un pezzo di casa al mio fianco, un punto riferimento, sempre presente nei tempi difficili e un grande amico con cui ho condiviso tanti momenti indimenticabili, al quale devo molto più di queste righe.

A João, un faro nel buio, sempre pronto ad aiutarmi e sostenermi come professionista e come amico, al quale devo molti dei miei traguardi.

Ai miei amici Rafa, Pedro ("Correia"), Nito, Cristiana, Marco, il tutto per i momenti indimenticabili. Che tanti altri vengano in futuro! Devo anche ringraziare i miei colleghi di laboratorio per l'aiuto, il buon umore e le distrazioni: a Rui, Diogo, Ana, Luis, Ivo, Sergio, per i tanti momenti insieme.

A tutti i miei amici Andréa, Paulo G., Eudis, Luciana, Suhely, Amanda, Gustavo, Marcelo, Gabi, Isabella, Karina, Tamara, per la loro amicizia, i momenti indimenticabili vissuti assieme e il loro aiuto in molti momenti difficili. Non avrei mai pensato di conoscere tanti brasiliani, mai ho pensato che il mondo potesse essere così sorprendente. A Heitor, Bruno, Hudson e le persone con cui ho condiviso il mio tempo a casa. A loro sono grato per la ricchezza che hanno portato nella mia vita.

Ai miei amici David, Sven ("o alemão"), Nuno L., Davi, Naaliel, Paulo V., Amir, Merk, con cui ho condiviso il mio percorso qui a Coimbra. A Naghmeh, amica fin dall'inizio di questa avventura del dottorato, a cui devo questo momento: senza quelle parole, nel 2012, probabilmente sarei in un luogo diverso in questo momento. A tutti i ragazzi del DEI con cui ho condiviso molti caffè, "Ph.D. dinner" e incontri veloci, pieni da risate e momenti divertenti, va la mia più profonda gratitudine.

A tutte le persone che ho incontrato in questi anni qui a Coimbra e in tutto il mondo, nella speranza di rivederli presto. A Christian, per i suoi consigli, il supporto e l'amicizia, e per tutti i momenti divertenti vissuti qui in Portogallo. A Fumio, Andreas e Andrea C., per il loro buon umore e la loro amicizia senza confini.

Vorrei esprimere la mia gratitudine alla mia famiglia, per il loro amore. Ringrazio i miei genitori Nunzio e Adele per i loro sacrifici e il loro continuo duro lavoro, che mi hanno dato la possibilità di attingere a questo obiettivo e di condividere con loro le meraviglie del mondo in cui viviamo. Li ringrazio per il loro amore incondizionato nei momenti belli e in quelli difficili, e per tutti i momenti che abbiamo vissuto insieme. Vi devo la mia vita.

A Simona e Dimitri, mia sorella e mio fratello, ai quali devo più di quanto possa esprimere, ringrazio per i momenti sempre buoni e divertenti, pieni di vita e di gioia. Non riesco a immaginare come sarebbe la mia vita senza di loro.

Ai miei cugini Moreno, Danilo e Francesca, per i tanti bei momenti, l'infinito supporto, e l'amore che ci rendono più che cugini.

A tutti i miei amici a casa, per il loro ininterrotto amore e sostegno, devo la mia parte più profonda, sperando di condividere tanti altri momenti in futuro.

Ringrazio Coimbra, per i tanti momenti ho passato qui e le lezioni apprese in questi anni. Ringrazio il Portogallo per essere diventata la mia seconda casa, per la gentilezza e accoglienza della sua gente, e per i momenti che sono stati e saranno.

Ringrazio tutto e tutti per aver procrastinato il mio lavoro, con un ringraziamento speciale al gruppo "Porta dos fundos" e molti altri YouTubers: senza loro non riesco a immaginare quel che sarebbe di noi. Ringrazio gli artisti per il loro lavoro costante ci mantiene umani. Possano sempre le distrazioni e la procrastinazione guidare e salvare le nostre vite.

Ora e sempre, carrega Benfica!

List of Publications

This thesis relies on the published scientific research presented in the following peer reviewed papers:

P 1. Marco Vieira, Ivano Irrera, Henrique Madeira, Miroslaw Malek, *"Fault injection for failure prediction methods validation"*, 2009, Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep 2009) at DSN 2009, Estoril, Lisbon, Portugal, 29 June - 2 July 2009 (DOI: *10.1109/DSN.2009.5270287*).

P 2. Ivano Irrera, João Durães, Marco Vieira, Henrique Madeira, *"Towards identifying the best variables for failure prediction using injection of realistic software faults"*, 2010, Proceedings of the 16th Pacific Rim International Symposium on Dependable Computing (PRDC'10), Tokyo, Japan, 13-15 December 2010, DOI: (*acceptance rate: 41,5%, 10.1109/PRDC.2010.51*).

P 3. Ivano Irrera, João Durães, Henrique Madeira, Marco Vieira, *"Assessing the Impact of Virtualization on the Generation of Failure Prediction Data"*, 2013, Proceedings of the 6th Latin-American Symposium on Dependable Computing (LADC'13), Rio de Janeiro, Brasil, 1-5 April 2013 (*acceptance rate: 38%,* DOI: *10.1109/LADC.2013.24*).

P 4. Ivano Irrera, Carlos Pereira, Marco Vieira, *"The Time Dimension in Predicting Failures: A Case Study"*, 2013, Proceedings of the 6th Latin-American Symposium on Dependable Computing (LADC'13), Rio de Janeiro, Brasil, 2013, 1-5 April 2013 (*acceptance rate: 38%,* DOI: *10.1109/LADC.2013.25*).

P 5. Ivano Irrera, João Durães, Marco Vieira, *"On the Need for Training Failure Prediction Algorithms in Evolving Software Systems"*, 2014, Proceedings of the 15th International Symposium on High-Assurance Systems Engineering (HASE'14), 2014, Miami, Florida, USA, 9-11 January 2014 (*acceptance rate: 30%,* DOI: *10.1109/HASE.2014.38*).

P 6. Ivano Irrera, Marco Vieira, *"A Practical Approach for Generating Failure Data for Assessing and Comparing Failure Prediction Algorithms"*, 2014, Proceedings of the IEEE 20th Pacific Rim International Symposium on Dependable Computing (PRDC'14), Singapore, 18-21 November 2014 (*acceptance rate: 37,5%*, DOI: *10.1109/PRDC.2014.19*).

P 7. Ivano Irrera, João Durães, Marco Vieira, *"Adaptive Failure Prediction for Computer Systems: A Framework and a Case Study"*, 2015, Proceedings of the 16th International Symposium on High-Assurance Systems Engineering (HASE'15), 2015, Daytona Beach, Florida, USA, 8-10 January 2015 (*acceptance rate: ?*, DOI: *10.1109/HASE.2014.38*).

P 8. Ivano Irrera, Marco Vieira, *"Towards assessing representativeness of fault injection-generated Failure Data for Online Failure Prediction"*, 2015, *to appear in* Proceeding of the 1st Workshop on Recent Advances in the DependabIlity AssessmeNt of Complex systEms (RADIANCE 2015) at DSN 2015, Rio de Janeiro, Brazil.

Other published works:

P 9. Valentina Bonfiglio, Leonardo Montecchi, Ivano Irrera, Francesco Rossi, Paolo Lollini, Andrea Bondavalli, *"Software Fault Emulation at model-level: towards Automated Software FMEA"*, 2015, *to appear in* Proceeding of the 1st Workshop on Safety and Security of Intelligent Vehicles (SSIV 2015) at DSN 2015, Rio de Janeiro, Brazil.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Computer systems are an intrinsic element of our life. Hardware and software components work jointly to provide complex functions, with the logic of the system embodied into the software, while the hardware provides data storing, data management, and communication. Despite the importance that hardware had over many decades, the software is nowadays the flexible and powerful mean that allows building and controlling increasingly complex systems. In fact, several areas of our society strictly depend on software systems, such as transportation (e.g., trains, airplanes, cars, elevators, etc.), industrial production, communication, medical aid (e.g., devices for intensive care, radiation therapy systems, etc.), and finance (e.g., banks, commercial systems, stock market).

Over the last decade, software has grown in complexity up to a point that systems are hardly free from defects (also known as *software faults*), and it is nowadays commonly accepted that every computer system eventually fails due to residual software faults, i.e., defects that escape the development and testing phases (Gray 1986; Ko, Dosono, and Duriseti 2014). A software fault can be a design flaw or a defect introduced during the coding of a software component (or the use of an external software) that is activated under certain conditions. Although hardware faults used to be the main cause of system outages (e.g., a bit flip in a hardware component due to a quantity of gamma rays higher than expected passing through the component), the situation has changed due to the higher reliability of hardware components and to the increasing size and complexity of software (Gray 1986; Lee and Iyer 1995). Starting from 1980's, studies point software faults as the major cause of computer failures (Kalyanakrishnam, Kalbarczyk, and Iyer 1999; Lee and Iyer 1995), and their weight with respect to hardware faults tends to increase, given the continuous growth of software complexity.

The problem is that, in complex systems, such as business- and mission-critical systems, software faults frequently lead to errors and failures of parts of the system (or of the system as a whole) that can ultimately lead to loss of profit or even human lives. This scenario called the attention of the scientific community to the way software systems are developed and to the properties that they must have to avoid hazards, as well as to techniques able to contain potential damages. In particular, the main interest has been on increasing the *trust* that one can put in the computer system or how much a user can *depend* on the system. Avizienis et al. define such concept as *system dependability* (Avizienis et al. 2004).

Several techniques were developed with the objective of avoiding or managing faults. A first family of techniques stands on the hypothesis that faults can be avoided. *Fault prevention* or *avoidance* techniques improve the software development process by applying, for instance, product quality controls and formal verification techniques. Likewise, *fault removal* techniques aim at finding and correcting the highest possible number of faults in a system after its development: examples are testing, verification, and validation techniques. On the other hand, it is well established that deploying fault-free complex systems is an unachievable goal (R. Chillarege 1995; Sullivan and Chillarege 1991; Ram Chillarege, Kao, and Condit 1991). This way, failures remain unavoidable events, and computer systems do need to encompass techniques for proactively handling and/or recovering from their effects. The second family of techniques, addressing such hypothesis, comprises *fault tolerance* that mitigates existing faults (e.g., by using system replication, error detection, system re-initialization, etc.), and *fault forecasting* for modeling the future impact of the faults present in the system, as well as the time when the system will fail (e.g., by using reliability block diagrams, fault-trees, etc.). The focus of this work is precisely on fault forecasting techniques, in particular on Online Failure Prediction, making use of past and current system data for estimating or forecasting failures.

Using *fault forecasting* techniques to predict if and when a failure would occur, allows applying countermeasures to avoid the occurrence of the failure itself, or at least preparing mechanisms in advance to recover from the failure. Such solutions can, for instance, reduce the time-to-repair and increase the availability of a system (F. Salfner and Malek 2005). In practice, estimations based on information about the failures faced by the system, such as Mean Time To Failure (MTTF) or Mean Time Between Failures (MTBF), can be used to predict the occurrence of failures in an interval of time. Also, reliability models (Lyu and others 1996), built using information about the development process, allow predicting future failures (assuming that the system properties are stable over time). However, the effectiveness of such forecasting methods is strongly limited, as the hypothesis that a given failure prediction model holds over time is nowadays proven invalid for most systems.

Online Failure Prediction (F. Salfner, Lenk, and Malek 2010) is a promising technique that improves the classical failure forecasting schema by correlating past data about the system state with the occurrence of failures, and afterwards comparing that information with the system state at runtime. The prediction information obtained at runtime allows rapidly and proactively taking countermeasures before the failure occurs, such as saving data and restart a failing system component, thus minimizing or even eliminating downtime and increasing availability (F. Salfner, Hoffmann, and Malek 2005). In practice, an online failure predictor forecasts incoming failures using *past* data from the system (used to train the predictor) and information about the *current state* of the system (obtained by monitoring system state variables), during the system execution (i.e., *online*). The information about the system state used can be *numerical variables* measuring relevant properties (memory available, page faults, etc.), or *categorical variables*, such as events from error logs (F. Salfner, Lenk, and Malek 2010). The output of online failure prediction can be either a decision that a failure is imminent, or some continuous measure that portrays how failure prone the current system state is. Preliminary estimates show that five minutes in advance failure prediction can improve system availability by an order of magnitude (G. Hoffmann and Malek 2006). This technique is particularly useful to address residual faults (i.e., faults that escaped the testing process) that cannot be tolerated by existing fault tolerance mechanisms (M. Vieira et al. 2009).

Several different types of online failure prediction algorithms were proposed in the past (see survey in (F. Salfner, Lenk, and Malek 2010)). An example is a method based on the system state clustering and Hidden Semi-Markov Models to detect failure-prone states, proposed by Salfner et al. (F. Salfner and Malek 2007). Another example is the use of Support Vector Machine classifiers to predict failures in hard disk drives presented by (G. F. Hughes et al. 2002). Online failure prediction has been proposed for single-node and multi-node systems, moving to cloud systems (Y. Watanabe et al. 2012) in the last years. There are also some examples of companies announcing failure prediction features in their systems (Liang et al. 2006).

Despite its potential contribution for improving dependability, online failure prediction still presents several limitations. In fact, in addition to the problem of choosing the optimal set of variables to use for prediction (the *feature selection* problem), online prediction models are difficult to tune and assess. Although these are common problems in the prediction field, in the failure prediction area they are exacerbated by the fact that the collection of failure data is extremely difficult and time-consuming, as demonstrated by several works that show that failure data collection can take from months up to years (Li, Vaidyanathan, and Trivedi 2002; G. Hoffmann and Malek 2006; Otsuka et al. 2014; Pitakrat, Van Hoorn, and Grunske 2014). Moreover, computer systems are dynamic and evolving in nature, and updating online failure predictors at regular intervals in absence of updated failure data is a difficult task. A proposed solution to address the problem of failure data scarcity is the use of failure data repositories (Felix Salfner, Lenk, and Malek 2010) provided in the form of collaborative databases where failure data coming from several (types of) systems are stored. However, although being a valid solution, failure data repositories must keep increasingly larger amounts of data associated to several particular systems, they take long to be built, and data become obsolete as systems change over time. Although proposals of failure prediction solutions adapting to such situation can be found in the literature, such approaches are limited to failure prediction models based on specific approaches (e.g., online failure prediction based on time series (Zemouri and Zerhouni 2011)).

1.1 Problem statement

In this work we argue that **advancing the failure prediction area requires a systematic approach that facilitates the generation of failure data**. Our proposal is to use realistic software fault injection to increase the probability of failures to occur, allowing speeding up the generation and collection of failure-related data. The thesis is that failure data generated by injecting residual software faults[1] can be used to support the process of training, assessing and comparing failure prediction models, as well as optimizing and promoting their use.

Fault injection consists of deliberately introducing faults in a way that emulates the existence of residual faults in the system (Arlat, Crouzet, and Laprie 1989), which may lead to errors (*erroneous system state*) and finally to system failures (*a deviation on the service provided*). Inoculating a system with software faults increases the number of faults in the system, which obviously increases the probability of the system to fail. In particular, injecting *realistic faults* means that the faults introduced represent defects that developers *could* potentially introduce during the system development. The ultimate idea is that injecting realistic software faults for increasing the probability of occurrence of failures enables the faster collection of failure-related data. Consequently, the availability of such data facilitates the assessment, comparison and improvement of existing failure prediction methods, or even the definition of new approaches.

Although this idea seems rather straightforward, fault injection techniques have been rarely used to improve failure prediction. Some works used the injection of memory leaks to accelerate the occurrence of the Software Aging phenomena in the system under study, but they targeted only aging-related failures (Gross, Bhardwaj, and Bickford 2002; Andrzejak, Moser, and Silva 2007; Alonso, Torres, and Gavaldà 2009). Furthermore, faults are mostly injected at the source code level, which is not practical when considering complex computer systems, and even not feasible when the source code is not available. Other techniques, such as executing stressful workloads (Vaidyanathan et al. 2001) that drive the operating system to limit

[1] In the fault injection context, the concept of injecting "residual" faults is used to highlight the fact that the injected faults do emulate residual software faults, i.e., faults that escaped the quality assurance activities conducted in the context of the software development project (e.g., testing phase), thus remaining in the system after deployment.

conditions (e.g., increasingly opening files, using large percentages of CPU cycles, etc.) (Magalhaes and Silva 2012), and accelerated life tests (Matias, Trivedi, and Maciel 2010), were proposed and used to raise failures and validate software rejuvenation models. In practice, such techniques exploit the faults already present in the target software system to speed up the occurrence of system failures.

In this work we hypothesize that the injection of realistic software faults is a valid solution for fostering the use of failure prediction, by addressing the following issues:

1. **Deploying online failure prediction models and evaluating their figures of merit on a particular system installation**. Effectively deploying failure prediction models requires accurate training, which can only be achieved by collecting failure data from a particular system installation, besides assessing the algorithms' performance and comparing them for selecting the most suitable for such system. In fact, studies on the performance of existing failure prediction models found in literature (a survey is in (F. Salfner, Lenk, and Malek 2010)) are often not comparable among each other (besides presenting frequently incomplete information), as they are very specific to the target system used in the work. Although public repositories for collecting failure datasets have become available in the last decade (e.g., the Computer Failure Data Repository (Usenix and Carnegie Mellon University (CMU) 2006)), the dependency between the public failure data and the system from which they were collected may hinder the training of predictors on a different system installation.
2. **Supporting the continuous adaptation of failure prediction in dynamic systems**. The characteristics of a complex system may change over time, due to a system update or a change of one or more hardware components (e.g., as in cloud-based systems). However, updating online failure predictors at regular intervals in absence of updated failure data is a hard task. Although some works addressed the problem of adapting failure prediction models over time, taking advantage of adaptive prediction models (Pitakrat, Van Hoorn, and Grunske 2014; Zemouri and Zerhouni 2011), there is still the need for failure data reflecting changes in the target system (e.g., new hardware components, software updates, workload variation, etc.).
3. **Identifying the best variables to be used to predict failures**. Online Failure Prediction models are based on runtime monitoring of parameters or *variables* that portray the system state. The selection of the system parameters to monitor is not trivial, as the number of variables can be very high and the ones to be used are not known *a priori*. Focusing on the best ones is essential to correctly use a predictor and to improve its performance. Moreover, as demonstrated by Hoffman (G.A. Hoffmann, Trivedi, and Malek 2007), obtaining an optimal online failure prediction model is more dependent on

selecting the right variables to support the prediction process, than on the choice of the model. The problem is that a large amount of field failure data is needed for identifying the variables that show the best symptoms of incoming failures.

1.2 Main contributions of the work

The objective of this work is to **advance the failure prediction state of the art by studying the applicability of realistic fault injection for generating failure data**, which can help training, assessing and comparing online failure prediction models on a concrete target system. In practice, the results achieved in this work open the door for the scientific community to address the actual problems in developing and deploying systems with failure prediction capabilities. In detail, the contributions of this work are:

1. **A conceptual framework for generating failure-related data making use of realistic software fault injection**. The framework is based on an experimental procedure in which realistic software fault injection campaigns are used for accelerating the occurrence of failures and thus generating failure-related data. The proposed framework should be implemented on the target system, thus software fault injection campaigns can provide extensive datasets representing realistic scenarios of the system execution (I. Irrera and Vieira 2014).
2. **An approach to study the accuracy of failure data generated using software fault injection**. When leading the system to fail through the injection of faults, one must be sure that the generated failure data are similar to the one likely to be collected in the operational scenario of the target system. We call this property *accuracy of the failure data*. Although the use of a realistic software fault injection technique contributes for achieving this property, such property needs to be validated, which unfortunately is not possible due to the already stated scarcity of failure data. This way, we propose an empirical approach for estimating the accuracy of generated (synthetic) failure data, based on the use of metrics (or estimators) that portray the correlation between the failure data generated by injecting faults and real failure data. The approach can be used for assuring a controlled and quality-driven data generation (I. Irrera and Vieira 2014).
3. **A study on the adoption of virtualization as a sandboxing environment for generating failure-related data.** As in most cases it is not possible to inject faults in a production system (especially when new failure data is periodically needed to accommodate system changes and evolution), we study the hypothesis of applying fault injection over an independent copy of the system. In particular, we investigate whether virtualization can be used

as a sandboxing solution for hosting such virtualized copy and study the applicability of such solution by assessing whether the data generated from the virtualized copy are adequate for training failure prediction mechanisms to be run in the original system. The correlation of failure data generated in a given system with data from several virtualized copies is analyzed based on the correlation of failure symptoms (Ivano Irrera et al. 2013).

4. **A conceptual framework for assessing and comparing alternative failure prediction models in the context of a particular target system.** We propose a conceptual benchmarking framework that can be instantiated to specific systems for a fair and sound assessment of alternative online failure prediction models. The framework makes use of the proposed approach for generating failure data and envisages the definition of the metrics, the workload to be used, as well as the process to implement the benchmark. This includes validating the properties of the benchmark, which increases confidence in the assessment and comparison of the results (I. Irrera and Vieira 2014).
5. **A conceptual framework for the automatic and continuous self-adaptation of failure prediction systems.** We define a generic framework (called Adaptive Failure Prediction – *AFP* – Framework) that can be instantiated to specific systems, whose goal is to train failure prediction models after the occurrence of specific events (e.g., a software update), collecting failure data when needed. The framework uses virtualization as a sandboxing environment for performing the fault injection process, taking into account the impact of the virtualization on the failure data generation. In practice, a replica executes a workload that mimics the operations executed in the original system to keep the failure data collected realistic. The failure predictors (re-)training process is automated, based on a modular event-driven architecture to detect when retraining is needed (Ivano Irrera, Vieira, and Duraes 2015).
6. **A study on system variables presenting failure symptoms to be used for predicting failures**. We study an approach to select the best variables for prediction purposes by identifying symptoms in a set of monitored variables, by generating failure data by fault injection and observing the impact of faults on the different variables that are being collected. The idea is that the activation of the injected faults may cause failure symptoms to show up in a limited group of variables, from which the most adequate for predicting failures should be selected. The impact is measured by applying a correlation function between each single variable and the failures observed in the system: the higher the correlation, the more likely the variable is to be suitable to predict failures (I. Irrera et al. 2010).

Besides the contributions stated above, another result of this work is the implementation of a **novel online failure prediction model**, which improves the failure prediction quality of a generic classifier-type predictor by including the time dimension in the prediction task (Ivano Irrera, Pereira, and Vieira 2013). Most online failure prediction techniques available nowadays make use of the current values of the monitored variables to perform the prediction, not considering the fact that the data used can be represented as time series (F. Salfner, Lenk, and Malek 2010). For instance, Hughes et al. (G. F. Hughes et al. 2002) apply SVMs (Support Vector Machines, (Cortes and Vapnik 1995)) to predict failures of hard disk drives, making use of several numerical variables (time series), but just one value at a time for each variable. Our idea is that the relation that the variables have with time (i.e., their continuous evolution) could improve the prediction quality of a model. This way, we propose the use of a *sliding window* applied to the available training data to improve the prediction performance of a SVM classifier (Ivano Irrera, Pereira, and Vieira 2013).

1.3 Structure of the document

This first chapter introduced the problem and the main contributions of the thesis.

Chapter 2 introduces basic concepts on dependability and fault tolerance, presents background and existing surveys on failure prediction models and algorithms, and discusses fault injection approaches and tools. Background on feature selection, computer benchmarking and virtualization is also presented.

Chapter 3 presents our approach for generating failure data by using realistic software fault injection. Besides describing the approach, we present a case study in which we use the generated data to assess the performance improvement of the sliding window technique applied to an SVM-based failure prediction algorithm. In addition, we present the method for the assessment of the accuracy of the generated failure data, which makes use of metrics estimating the distance between different failure datasets.

Chapter 4 studies the applicability of virtualization as a sandboxing solution for running a copy of the target system that can be used to generate failure-related data when injecting faults in the target system is not possible. In a case study we analyze the impact of virtualization on the generation of the data, and propose an approach for taking such factor into account when adopting this solution.

Chapter 5 presents the benchmarking framework for assessing and comparing alternative failure prediction models in the context of a particular target system. A case study demonstrates its applicability, efficacy and ease of installation, among other fundamental properties that a benchmark must have.

In Chapter 6 we propose a framework for supporting adaptive failure prediction, which is able to address the changes that occur in the system by automatically re-training and optimizing the failure prediction model. We also present a case study in which we implement and validate such framework.

Chapter 7 presents a study of a preliminary approach for selecting the best variables for prediction purposes. The goal is to identify the variables that show more symptoms of failures by correlating their values with the effectively observed failures. The case study presented shows the applicability of the proposed methods and confirms that only a subset of all the variables monitored should be used for failure prediction.

Finally, Chapter 8 concludes the thesis, summarizing the lessons learned, evidencing the potential of the proposed solutions, and presenting the weaknesses that we believe should be tackled as future work.

Chapter 2
Background and related work

In this chapter we present background concepts and related work on dependable systems (Section 2.1) and Online Failure Prediction (Sections 2.2), the use of virtualization solutions in the failure prediction context (Section 2.3), fault injection (Section 2.4) and its use for online failure prediction (Section 2.5), and computer systems benchmarking (Section 2.6). In particular, the chapter also includes an overview on the performance evaluation of failure prediction models (Section 2.2.4) and the prediction optimization problem, including feature selection (Section 2.2.5). Section 2.6 concludes the chapter.

2.1 Dependable computing

Computer systems naturally tend to grow in complexity up to a point where their behavior is partially unpredictable, especially for what concerns its software, which represents a major threat to the benefits they aim to provide.

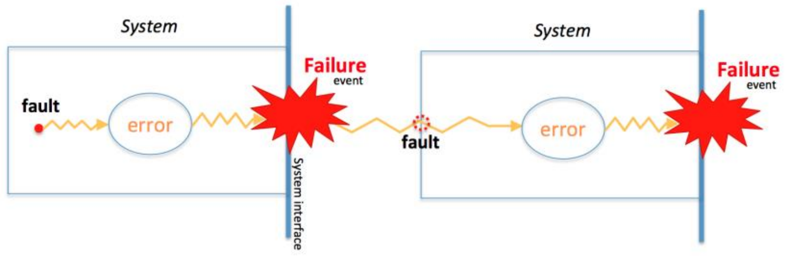
<<Nothing can assure the absence of errors>> (*E. W. Dijkstra*).

Demonstrating the *absence* of defects in the hardware and software components of a system before deployment (e.g., by using mathematical proofs) or after development (e.g., using testing) is a NP-complete and NP-hard problem (Cook 1971). In particular, *a posteriori* techniques can only assure the presence of faults, while preventive methods are used to decrease the presence of faults, hence only partially solving the problem. It is thus well known that complex computer systems do contain defects, and eventually fail (Gray 1986; Sauer 1993; Oppenheimer 2003).

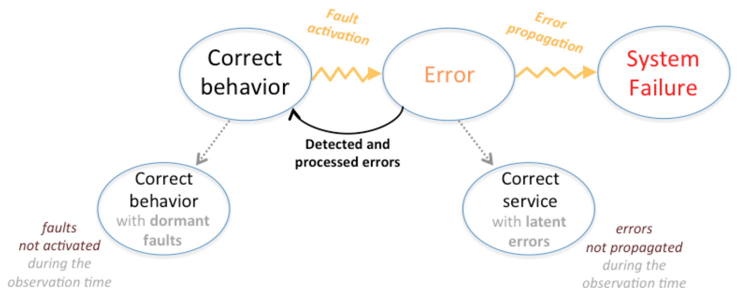
According to (Avizienis et al. 2004), a computer system generally provides a **service or functionality** that can be used by a human user or by another system. The system is working properly if it is able to provide the expected functionality in the correct way. The service that the system provides is a sequence of states perceivable at the system interface (*external states*) (Avizienis et al. 2004).

A deviation of the service delivered by the system from the expected service is called a **system failure**. The system is said to contain an **error**, or being in an erroneous state, when it state deviates from what is defined to be the correct one. However, even if the system is not in correct state, as long as it provides the expected service or functionality, there is no failure. Errors evolve into failures when the malfunction reaches the system interface, or is detected by an external user (e.g., a user cannot access its data, or the server cannot send to a user some requested data). Errors can accumulate without influencing the system service, or they may influence the system causing a **partial failure** or leading the system to run in a **degraded mode**. The adjudged or hypothesized cause of an *error* (and thus, a *failure*) is a **fault**, which can be internal (originating inside the system boundaries) or external (originating externally to the system, and that propagates into the system by interaction or interference). Faults can be of different kinds (e.g., hardware or software, transient or permanent). An exhaustive taxonomy can be found in (Avizienis et al. 2004).

Figure 2.1 (a) shows the **fault-error-failure** chain expressing the causality relationship between faults, errors and failures. It is worth noting that a failure of a component *can be* a fault (in the figure an external fault, as a voltage peak or an incorrect input) of a connected component. Figure 2.1 (b) shows the transition of system states in the presence of a fault. A system working correctly may contain faults that remain *dormant*, or that may be activated causing an *error*. Errors may remain latent and let the system continue to provide a correct service during the observation time, or propagate to the system interface, causing a *failure*.



(a) Fault-error-failure-fault chain


Correct behavior
Fault activation
Error
Error propagation
System Failure
Detected and processed errors
Correct behavior with dormant faults
faults not activated during the observation time
Correct service with latent errors
errors not propagated during the observation time

(b) System states in the presence of a fault

Figure 2.1 – The causality relationships between faults, errors and failures

The use of computer systems cannot be separated from the management of errors and failures, which can occur during its operations due to several and often unknown reasons, raising the need for techniques for assuring the correct system behavior, also in presence of unexpected events. This is especially critical for systems used in environments where failures may cost loss of business and human lives.

Computer-based systems can encompass specific properties addressing undesired errors and failure events. For instance, a *safe* system is a system that assures no major damages when a failure occurs, while a *reliable* system assures the continuity of its service. Such properties can be grouped under the generic concept of **dependability**: a *dependable* system is a system one can depend on. The authors of the work entitled "*Basic Concepts and taxonomy of Dependable and Secure Computing*" (Avizienis et al. 2004) give the following *definition* of dependability:

> *The* ***dependence*** *of system* ***A on*** *system* ***B*** *(...) represents the extent to which system* ***A's dependability is*** *(or would be)* ***affected*** *by that of System* ***B****. The concept of dependence leads to that of* ***trust****, which can very conveniently be defined as* ***accepted dependence****.*
>
> *As developed over the past three decades, dependability is an integrating concept that encompasses the following attributes:*
>
> - ***availability****: readiness for correct service.*
> - ***reliability****: continuity of correct service.*
> - ***safety****: absence of catastrophic consequences on the user(s) and the environment.*
> - ***integrity****: absence of improper system alterations.*
> - ***maintainability****: ability to undergo modifications and repairs.*

The intended **dependability** of a computer system (based on hardware and software) is defined by attributes in terms of the frequency and severity of the acceptable failures, for specified classes of faults and a given user environment (Avizienis et al. 2004). For example, the dependability attributes reliability and availability can be expressed mathematically as follows:

- **Reliability**: it can be modeled as the conditional probability of delivering a *correct service* in the interval *[0, t]*, given that the service is correct at the time *t=0*. The reliability therefore models the mission time of the system or the *time to failure T* (or *TTF*):

(2.1) $$R(0,t) = P(no\,failures\,in\,[0,t]|correct\,service\,in\,0)$$

The reliability function can thus be obtained considering the *failure occurrence time*. Hence, be T the failure time (or *time to failure, TTF*) and *$F_T(t)$* the cumulative distribution function (CDF) of the failure occurrence (or arrival) times, the reliability function can be expressed as:

(2.2) $$R(t) = \mathrm{P(T > t) = 1 - P(t < T) = 1 - F}_T\mathrm{(t)}$$

Moreover, assuming *T* (or *TTF*) as a random variable to be continuous positively defined, and *$F_T(t)$* to be differentiable, the CDF can be written as:

(2.3) $$F(t) = \mathrm{P(T < t)} = \int_0^t f(x)dx$$

for t > 0

Examples of *reliability functions R(t)* are obtained using exponential failure arrival times, Weibull distribution, Lognormal distribution, etc.

- **Availability**: a system is available at time *t* if it is able to provide a *correct service* at that instant of time. Thus:

(2.4) $$A(t) = \begin{cases} 1 & if\,correct\,service\,at\,t \\ 0 & otherwise \end{cases}$$

The availability can thus be modeled as the expected value *E[A(t)]*. In particular, if the time to failure is characterized by its mean, called *Mean Time To Failure* (*MTTF*), and the time to repair as well (*Mean Time To Repair, MTTR*), a function of the probability of finding the system in a correct state can be given by the rate *A* defined as:

(2.5) $$A = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF}$$

where *MTBF* is the *Mean Time Between Failures*.

Attaining dependability attributes requires systematic approaches aimed at improving the system quality in terms of hardware and software development, tolerance to errors and failure occurrence, verification of the correct functioning, validation of the functionalities and the mission of the system, and so on. According to (Avizienis et al. 2004), the techniques for assuring the dependability of a system can be divided into four categories, focusing on the concept of fault as the cause of errors and failures:

1) **Fault Prevention (or fault avoidance) techniques** that reduce the number of faults present in a computer system (i.e., preventing the existence of faults). Some examples are the improvement of the development processes of both hardware (e.g., avoiding bad design rules) and software (e.g., information hiding, modularization, use of strongly-typed programming languages). Although some faults are still present at the end, the use of such techniques is a necessary step towards dependable systems.
2) **Fault Tolerance techniques**, based on the hypothesis that faults do exist in the system and eventually will be activated causing errors or failures. Such techniques are aimed at avoiding failures (e.g., using back-up components for allowing the system to continue providing its service, or having redundant systems using diverse hardware and/or software elements), mitigating the effects of failures (e.g., planning the system to offer a degraded service when a component fails), or planning system recovery (e.g., reducing downtime and time to repair). Besides reacting to failures, fault tolerance techniques can be based on error detection and handling (e.g., roll-back or roll-forward to error-free states, or masking the error by using redundancy), and fault detection (e.g., the effect of memory leaks, resulting in software aging and system failing due to memory exhaustion, can be handled by rebooting the system from time to time).
3) **Fault Removal techniques**, for removing the faults either during the system development (e.g., verification, validation) or during its use. In particular, the removal of faults during the use of the system can be implemented as corrective maintenance (faults show themselves and are corrected) or preventive maintenance (uncovering existing faults before they evolve into errors).
4) **Fault Forecasting techniques**, which analyze the behavior of the system in a qualitative or quantitative way, with respect to the fault occurrence or activation. In practice, the aim is to evaluate the system behavior in order to estimate the future consequences of a fault (using system modeling or system testing). An example are reliability growth models (F. Salfner, Lenk, and Malek 2010), which are based on data about past failures (and thus activated faults) to model the time to failure. Worth to put on evidence are the dependability benchmarking approaches, whose goal is to assess measures of the behavior of systems in the presence of faults, whose results can be used by forecasting techniques (Avizienis et al. 2004).

In addition to the techniques and terms above, other concepts are used in the dependability context, including:

- **Resilience**, the property of a system to deliver a justifiably trusted service in a persistent way, when the system faces *changes* (Laprie 2005; Simoncini 2009). The definition of changes ranges from unexpected failures, attacks or accidents, to changes relative to the system load and configuration (Trivedi, Kim, and Ghosh 2009). In general, a resilient system is a system that is trustworthy and tolerant to changes falling outside the design envelope (Trivedi, Kim, and Ghosh 2009), while a dependable system deals with events inside the design envelope.
- **Assurance**, a measure of confidence that the features, practices, procedures and architecture of an information system accurately mediate and enforce safety and security policies (partially from (Committee on National Security Systems 2010), (SAFECode - Software Assurance Forum for Excellence in Code 2008), (NASA)). High-assurance systems must provide an high measure of confidence in the techniques used to attain dependability and/or security, in a way that gives "*justifiable trustworthiness in meeting established business and security objectives*" (OMG System Assurance Task Force). The concept of system assurance is strongly based on evidence, which can be obtained by measurement or formal methods. In practice, a high-assurance system is required to be safe and/or secure within well-defined limits and with a well-defined confidence (e.g., assure safety-critical systems to be safe).
- **Antifragility**, a novel concept based on the definition of fragility of a system (economical, financial, software, etc.) (Taleb 2012). A software system is *fragile* (Monperrus 2014) if during its development an error by omission (something missing, few things implemented) or commission (too many things to do) occurred. In particular, errors by omission mean that "*there may exist neglected design principles and implementation*", which drive to fragility (Monperrus 2014). The elimination of such neglected principles may lead to defining antifragile principles (Monperrus 2014). Preliminary work presented a comparison between the concept of antifragility and existing concepts (e.g., fault tolerance, robustness), and showed how automated runtime bug fixing and fault injection during production are two means to achieve antifragility.

2.2 Online Failure Prediction

Online Failure Prediction is a technique that allows forecasting failures occurring in a near future by monitoring the system at runtime (thus the term *online*), and using past information about the system's (normal or faulty) behavior (see Figure 2.2). A formalization of the Online Failure Prediction problem was proposed by Salfner et al. in (F. Salfner, Lenk, and Malek 2010). The output of online failure prediction can be either a decision that a failure is imminent, or some continuous measure that portrays how failure prone the current system state is. The prediction of failures is thus based on different kinds of information, including the *past* data from the system (used to train the predictor), the *current* information about the state of the system (obtained by monitoring system variables, using error reports, etc.), the *time horizon* of the prediction, among others. Online Failure Prediction involves techniques like machine learning, statistical analysis, pattern recognition, and so on. Failure predictors are normally trained and tuned in advance for a given target system using data related to failure events for that particular system. Predicting failures in advance allows avoiding failures or at least mitigating their effects (e.g., by saving data or preventively restarting specific system modules).

Online Failure Prediction is considered the natural evolution of **Failure Prediction** (also referred to as reliability modeling or prediction), which can be dated back to Nassar in 1985 (Nassar and Andrews 1985), and whose models rely only on historical failure data, i.e., information on failures occurred in the past. Such technique allows estimating reliability indicators, as MTTF (Mean Time To Failure) and MTBF (Mean Time Between Failures). However, the limitation of failure prediction stands in the fact that no information about the actual state of the system is taken into account, thus not achieving a precise and flexible prediction.

Many works on Failure Prediction integrating the information on the actual state of the system have appeared in the last 15 years. In fact, the very first work on predicting failures using the actual system state can be dated back to Wolski et al. (Wolski, Spring, and Peterson 1997) in 1997, which developed a system for predicting at runtime the performance of a network, for optimal resource allocation and scheduling decision for meta-computer systems[2]. Garg et al. (Garg et al. 1998), on the other hand, proposed a model for the estimation of resource exhaustion on an Apache web server in 1998, based on the slope estimation of a set of continuous numerical system parameters (i.e., used swap space, file table size, etc.) monitored from the system. In particular, the method was able to predict performance failures, as well as software aging-related failures (i.e., failures due to resource exhaustion based on the runtime system information) (Vaidyanathan and Trivedi 2001). However, not only numerical sequences were used in the past to predict failures: Lin and Siewiorek (Lin and Siewiorek 1990) developed a prediction technique, called DFT (Dispersion Frame Technique), based on the analysis of discrete events, rather than on numerical sequences. A fundamental difference between classic Reliability Prediction and the new types of prediction models stands in the fact that that the latter are able to predict failures likely to happen in a *short-term* (i.e., in some

[2] The system was called *"Network Weather Service"* and was based its predictions on linear models (like linear regression models), using data coming from sensors located in different elements of the network (nodes, links, etc.).

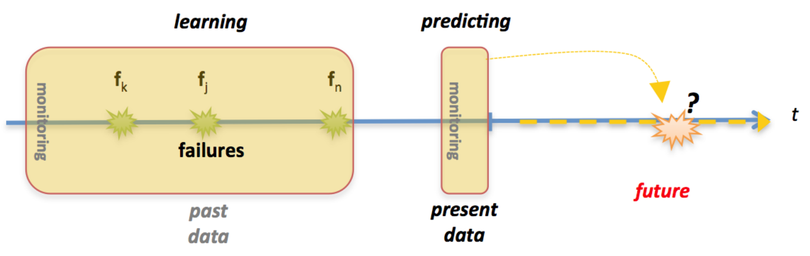


Figure 2.2 – Online Failure Prediction: an overview

minutes, hours, or days, depending on the scale of the system), which make them more suitable to model the actual life-time of a computer system.

In this thesis we use both *Failure Prediction* and *Online Failure Prediction* to refer to the techniques that predict incoming failures in a short-term based on both the past and the actual state information of the system under analysis. Moreover, when referring to Failure Prediction in the classical acceptation, we use the term *Reliability Prediction*, or similar. In the next subsections we introduce the main definitions in Online Failure Prediction, including its goal and employment in improving system dependability attributes. We also present a taxonomy of the existing Online Failure Prediction methods (as proposed by Salfner et al. in (F. Salfner, Lenk, and Malek 2010)), giving some insight on the most representative works that can be found in the literature.

2.2.1 The Online Failure Prediction context

Online Failure Prediction is particularly useful to address residual faults (i.e., faults that escaped the testing process) that cannot be tolerated by existing fault tolerance mechanisms, and thus are likely to evolve into failures. In a dependability context, as defined by (Avizienis et al. 2004), Online Failure Prediction can be considered a fault forecasting technique (being an evolution of reliability growth models, as mentioned before), and an enabler for fault tolerant systems. In fact, predicting failures may not be enough to achieve perfect dependability, but the information about an incoming failure can be used to prevent failures from occurring on the system, or at least to prepare the recovery mechanisms with some time in advance, thus reducing the time to repair.

The authors of (F. Salfner, Lenk, and Malek 2010) propose the use of Online Failure Prediction in a broader framework, called **Proactive Fault Tolerance** (PFT), in which the prediction of failures is the starting step towards their management or the management of their effects. In particular, the Proactive Fault Management (Figure 2.3) is based on monitoring the system (also considering the problem of feature selection, which may change over time), evaluating the system state through the use of Online Failure Prediction (as well as the evaluation of the system for diagnosing

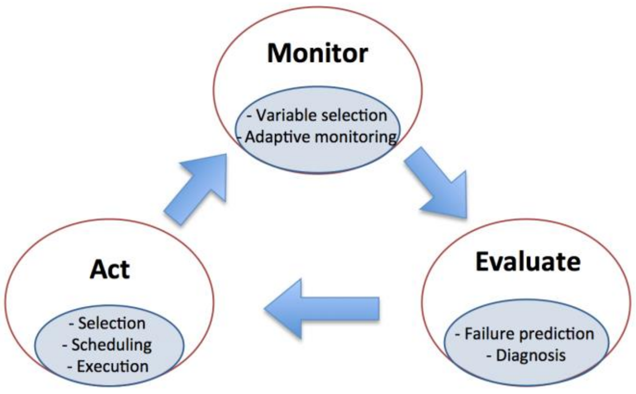


Figure 2.3 – The MEA (Monitor-Evaluate-Act) Proactive Fault Tolerance schema

for existing faults), and acting accordingly (in which one can select, schedule or execute several types of tasks).

The use of failure prediction in a reactive schema can be a way to limit downtime or even avoid a failure, e.g., through the use of fault tolerance, fault removal, early-recovery mechanisms, and so on. In fact, in the worst case, if a *predicted* failure cannot be avoided, one can use the information about it for anticipating recovery duties, allowing the reduction of the relative system downtime (see Figure 2.4). In case a failure is not avoidable, *anticipating repairing duties* implies a reduction of the Mean Time to Repair (MTTR), which corresponds to a growth of the system availability, as:

$$A = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF} \qquad (2.6)$$

while the reliability is not affected, as its definition only depends on the failure occurrence in an interval *T*:

$$R(t) = \mathrm{P}(\mathrm{T} > \mathrm{t}) = 1 - \mathrm{P}(\mathrm{T} < \mathrm{t}) = 1 - \mathrm{F}_T(\mathrm{t}) = 1 - \int_0^t f(x)dx \qquad (2.7)$$

On the other hand, *avoiding* failures by using information from Online Failure Prediction can improve both the reliability and the availability of the system, as the Mean Time To Failure (MTTF) increases both at the numerator and the denominator of equation (2.6), letting the availability tend to 1.

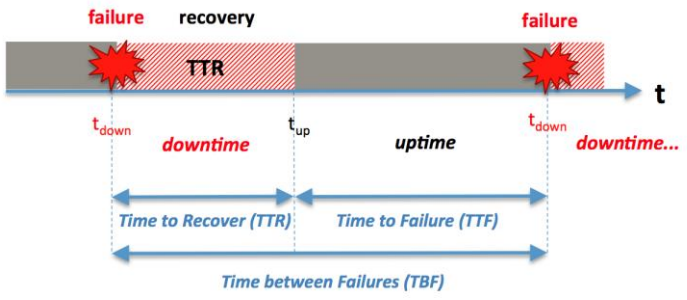


Figure 2.4 – Relations between failures and system down- and up-time

2.2.2 A taxonomy and examples of online failure prediction methods

(F. Salfner, Lenk, and Malek 2010) presents a comprehensive taxonomy of the manifold of online failure prediction approaches for computer-based and generic systems. The **taxonomy** is based on the fault-error-failure model: the authors consider that the system state can evolve into a failure through four stages: fault, undetected error, detected error, and failure. An *error remains undetected* until an *error detector* identifies the incorrect state (see Figure 2.1 (b)). The authors found that a great part of the surveyed failure prediction systems are based on the concept of *failure symptoms* (e.g., aging trends in Software Aging detection). For this reason, their taxonomy is extended with the *failure symptoms* dimension: besides causing a failure, an error (detected or undetected) can cause out-of-norm behavior of system parameters as a side effect (F. Salfner, Lenk, and Malek 2010). Figure 2.5 shows the relation between the fault-error-failure model and the existing online failure prediction approaches. As the authors of the survey propose, each online failure prediction approach can be categorized based on the kind of input data used.

The authors surveyed over 50 different approaches among the most known monitoring-based failure prediction approaches existing in the dependability prediction literature. The complete taxonomy is presented in Figure 2.6.

The existing online failure prediction approaches can be thus distinguished in four categories (see Figure 2.6): **failure-tracking** techniques, **symptoms monitoring** techniques, **detected errors reporting** mechanisms, and **undetected error auditing**. Except for failure tracking (included in the survey only for coherence purposes, as it does not perform failure prediction "at runtime", but uses only information from the past), all the remaining techniques use information about the intermediate stages of the fault activation to infer failure-prone situations, thus trying to act in advance.

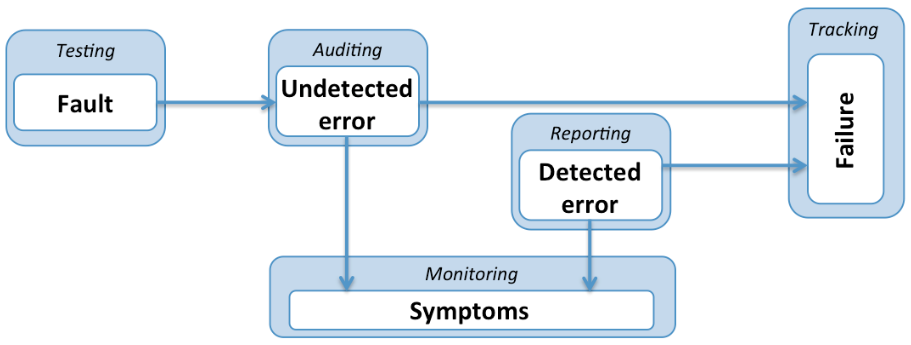


Figure 2.5 – Relation between the fault-error-failure model and the approaches for Online Failure Prediction at the state of the art (F. Salfner, Lenk, and Malek 2010)

According to Salfner's survey the taxonomy can be navigated as follows (F. Salfner, Lenk, and Malek 2010):

1) **Undetected errors auditing.** Auditing can identify undetected errors, and includes techniques that check whether the entity under audit is in an incorrect state. For example, memory auditing would inspect used data structures by check-summing. However, failure prediction literature presents no works that can be contained in this dimension, which may be due to the fact that prediction based on auditing results may drive to low prediction quality.

2) **Detected errors reporting.** Once an error detector identifies an incorrect state the detected error may become visible by reporting. Reports are written to some logging mechanism such as log-files or Simple Network Management Protocol (SNMP) messages. Techniques used for predicting failures based on error reports include classifiers, rule-based approaches, pattern recognition,

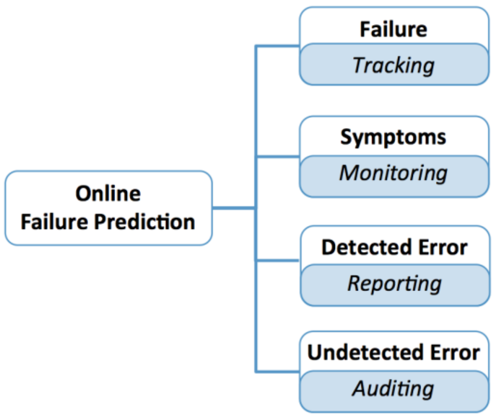


Figure 2.6 – A taxonomy of existing Failure Prediction approaches (F. Salfner, Lenk, and Malek 2010)

and statistical tests.

3) **Symptoms monitoring.** Symptoms are side effects of errors that can be identified by monitoring system parameters such as memory usage, workload, sequence of function calls, etc. An undetected error can be made visible by identifying *out-of-norm behavior* of the monitored system variable(s). Techniques used in this direction for predicting failures include function approximation, classifiers, system modeling, and time series analysis.

4) **Failures tracking.** The occurrence of failures can be made visible by tracking mechanisms. Tracking includes, for example, watching service response times or sending testing requests to the system for the purpose of monitoring. Techniques used for failure tracking are mainly failure co-occurrence analysis and probability distribution estimation.

In the next subsection we briefly describe two of these categories through some example of works found in literature, thus for providing a wider and complete view of the failure prediction scenario. We focus on the categories Detected Error Reporting and Symptoms Monitoring, as they represent most of the works in the failure prediction area.

2.2.2.1 Failure prediction based on detected error reporting

An error occurs when a fault is activated, i.e., the fault brings the system in an unpredicted and erroneous state, and the detection event is usually reported using some system logging facility. The category of failure prediction approaches that use the information on errors deal with event-driven approaches, working on discrete events (i.e., the *categorical data* obtained) based on periodic observations.

One of the most used approaches to understand if a failure is going to happen is to evaluate if a well-known set of conditions is met. If so, the situation is judged failure-prone. As pointed out by Salfner et al. in (F. Salfner, Lenk, and Malek 2010), usually the rule-based failure prediction has the following simple form:

IF <condition1> THEN <failure warning>
IF <condition2> THEN <failure warning>
…

Hence, the goal of failure prediction algorithms in this category is to identify, in an automatic way, the conditions algorithmically from a set of training data. As an example, Hätönen et al. (Hatonen et al. 1996) described a system that identifies episode rules (alarms) from error logs: these rules are in the form "if errors A and B occur within 5 seconds, then error C occurs within 30 seconds with probability 0.8". Weiss introduced a failure prediction technique called "*Timeweaver*", which is based on a genetic training algorithm (Weiss 1999). Other methods analyze errors that occur close together either in time or in space (on the hypothesis that errors spread in time or space before a system failure), considering their distribution similarly to the case of previous failures occurrence. To this group belong works like Levy and Chillarege (Levy and Chillarege 2003), and Lin and Siewiorek (Lin and Siewiorek 1990). In the latter, the dispersion frame technique (DFT) can be considered belonging to this family of approaches as it also uses a set of heuristic rules on the time of occurrence of consecutive error events of each component to identify looming permanent failures.

Some works used *pattern recognition* and *statistical tests* to address the problem of predicting failures using error reports. Also *classifiers* were used with this type of discrete information. An example is the DFT technique by Lin and Siewiorek (Lin and Siewiorek 1990), which focus on the time when errors are detected and uses pattern recognition to identify failure prone situations based on the time relations of the error events. Similarly, Salfner et al. (F. Salfner and Malek 2007; F. Salfner 2006) presented Similar Events Prediction, a technique based on a semi Markov chain model and Hidden Semi Markov models (HSMM), which is more flexible with respect to the former method. These methods are able to identify patterns that indicate an upcoming failure: in particular, a ranking value is assigned to an observed sequence of error reports expressing similarity to patterns that are known to lead to system failures and to patterns that are known not to lead to system failures. The final prediction is then accomplished by classification based on similarity rankings. An example of classifiers applied to discrete data (error events, in this case) can be found in (Domeniconi et al. 2002), which used an approach based on SVM and SVD (Single Value Decomposition) to classify if a situation is failure-prone or not. As also stated by Salfner in his survey (F. Salfner, Lenk, and Malek 2010), the main difficulty that this kind of approaches can face is that generally one single detected error is not sufficient to infer if a failure is going to occur or not. This way, the input data vector is usually obtained from several errors reported within a time window.

2.2.2.2 Failure prediction based on symptoms monitoring

Several methods were developed in the last years taking advantage of the analysis of *contour* information about the system. Failure prediction methods belonging to this class are able to detect **symptoms** of an upcoming failure based on the continuous analysis of monitored data. As introduced above, symptoms are side effects of errors that can be observed by identifying *out-of-norm behavior* from monitored system variables.

Works on *software aging*[3] and *rejuvenation*[4] found in literature belong to this class of online failure prediction algorithms. Besides, some of these works present a complete fault management schema, under the concept of Software Rejuvenation. The first interesting work in this field is (Vaidyanathan and Trivedi 1999), where the authors tried to dynamically assess the optimal time to restart of an Apache web server suffering from Software Aging. The idea is to approximate the amount of swap space used and the amount of real free memory (target functions) using a semi-Markov reward model for estimating the exhaustion of system resources as a function of the workload being executed and the execution time. Li et al. (Li, Vaidyanathan, and Trivedi 2002) developed a model using regression ARX models: 7 univariate (MISO) ARX models were built for each predicted variable (e.g., "PhysicalMemoryFree", "SwapSpaceUsed", etc.), and then combined into a single multivariate model (MIMO ARX).

Hoffmann (G.A. Hoffmann, Salfner, and Malek 2004; G.A. Hoffmann, Trivedi, and Malek 2007) proposed a prediction model called UBF (Universal Basis Functions), a generalization of the kernel functions of the well known Radial Basis functions (RBF) technique. In this work the authors also compare several prediction methods, among which ARX, UBF, RBF, and SVM. Other methods used neural networks, like for instance the one from Fu and Xu (Fu and Xu 2007), which use a neural network to approximate the number of failures in a given interval.

The methods mentioned above made use of *function approximation* techniques, trying to infer the unknown functional relationship between monitored system variables and a target value (for instance the "used swap space" in (Vaidyanathan and Trivedi 1999)). Other techniques found in literature are based on time series analysis (similar to function approximation), classifiers, system models, and so on. Another example is the work of Garg et al. (Garg et al. 1998) that used regression models on measured system variables "real memory free", "size of file table", "process table size", and "used swap space" of UNIX machines for estimating the resource exhaustion, and thus a possible future failure.

Other failure prediction algorithms evaluate the current value of system variables directly, instead of approximating a target function, or analyzing several successive samples of a system variable, and the current situation (at each time *t*) is classified as failure-prone or *not*. The *classifier* decision boundary is derived from past system behavior, where the system failed or not. In this case, Online Failure Prediction is performed at runtime for checking on which side of the decision boundary the current monitoring values are. The data used by the classifiers can be nominal (e.g.,

[3] *Software aging* is a phenomenon for which the state of the system degrades over time, eventually leading the system to fail.

[4] *Software rejuvenation* is a proactive technique that aims at reducing the probability of future failures due to software aging. In practice, it is a Failure Prediction technique.

events) or numerical. An interesting example can be found in the work of Murray et al. (Murray, Hughes, and Kreutz-Delgado 2003), which used SVMs to predict failures on hard disk drives. Some other works used Bayesian failure prediction approaches for solving the prediction problem, like for instance Bodik et al. (Bodik et al. 2005), where the hit frequencies of a big commercial website were analyzed in order to identify non fail-stop failures, using a naïve Bayes classifier.

Online Failure Prediction approaches may also be based only on a model of the normal system behavior, i.e., *failure free* (in contrast with the classifier approach that requires training data for both the failure-prone and non failure-prone case). In this case, at runtime, the current measured system behavior is compared to the expected normal behavior, and a failure is predicted in case of deviation.

2.2.3 The Failure Prediction problem: a model and its parameters

An online failure predictor forecasts incoming failures at runtime, based on past data from the system (used to train the predictor), and information about the current state of the system (obtained by monitoring system variables), among others. The information used can be numerical, such as variables measuring properties of the system (free memory, page faults, etc.), or categorical, such as events from error logs.

A model for characterizing the **online failure prediction problem** was proposed by Salfner et al. in (F. Salfner, Lenk, and Malek 2010). The failure prediction task consists of assessing if, at a time t, a failure is going to occur within a precise time, called *lead-time* Δt_l. The prediction can be valid in a given time window, called *prediction window* Δt_p. The variation of the parameters Δt_l and Δt_p influences the performance of the prediction. In practice, at time t, a model (or predictor) should predict if a failure is going to occur in the interval $[t+\Delta t_l,\ t+\Delta t_l+\Delta t_p]$. As shown in Figure 2.7, a prediction performed at time t targets the *Prediction Window* starting at time $t+\Delta t_l$, and lasting Δt_p.

The prediction can be valid until $t+\Delta t_l+\Delta t_p$. As mentioned before, the predictor is built from a set of past data. Taking the definitions from (F. Salfner and Malek 2010), considering a classifier as prediction system, we can assume that these data are a set of observations $x=<f_1, f_2, \ldots, f_n>$ of a target system. The *prediction task* is then to predict, from the observed features $x_{new} =<f_1, f_2, \ldots, f_{n-1}, ?>$, the target variable f_n, which

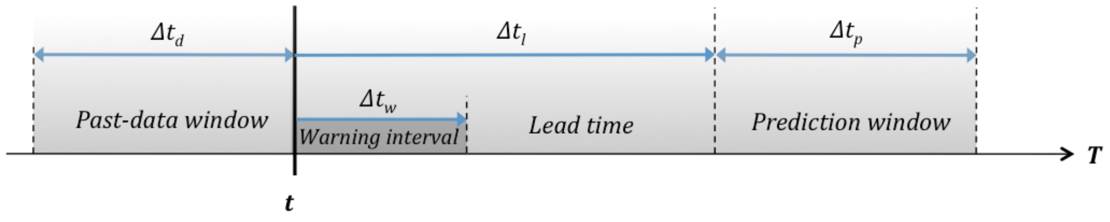


Figure 2.7 – Time relations in Online Failure Prediction

can be either "failure" or "no failure" or, in general, a continuous measure indicating how much failure prone the current system state is. Thus, given previously unseen observation matrix x_{new} with an unknown class label at time t, the *prediction about the occurrence of a failure in the interval* $[t+\Delta t_l, t+\Delta t_l+\Delta t_p]$ is given by $f_n=C_l(x_{new})$, where C_l is the predictor. In particular, a prediction at time t is correct if the target event occurs at least once within the prediction period Δt_p.

Varying the prediction parameters influences the accuracy and efficacy of the predictor, the computational power needed to perform the prediction task, among others. The prediction is based on the current system state, which is assessed by system monitoring within a data window of length Δt_d and the prediction period Δt_p defines how far the prediction extends into the future, which depends on the problem domain (e.g., how long it takes to restart a component, how long it takes to initiate a failover sequence, how much complex the system is, etc.). Increasing Δt_p increases the probability of a failure to be predicted correctly. On the other hand, if Δt_p is too large, the prediction is of little use since it is not clear when exactly the failure will occur. The lead-time At_l covers the time frame in which the prediction is valid, and it is necessary for a prediction to be of any use. Since failure prediction does not make sense if the lead-time is larger than the time the system needs to react in order to avoid a failure or to prepare for it, (F. Salfner, Lenk, and Malek 2010) introduce the minimal warning time Δt_w (see Figure 2.7). If the lead-time is shorter than the warning time, then there is not enough time to perform any preparatory or preventive actions.

2.2.4 Performance evaluation of Failure Prediction models

Evaluating the figures of merit of a single prediction system and comparing different prediction models are open problems in the online failure prediction context.

(F. Salfner, Lenk, and Malek 2010) presents a performance evaluation schema based on the model for the failure prediction problem proposed by the same authors. A prediction performed at time t produces a single prediction value: the prediction is correct if the failure event occurs at least once within the prediction interval. In such case, a True Positive is obtained (TP). If no failure occurs, a False Positive (FP) is obtained. The dual cases are True Negative (TN) and False Negative (FN). All the possible cases are represented in Figure 2.8 in the so-called *contingency table*. A particular instantiation of a contingency table is called *confusion matrix*.

A model can predict once or several times in a certain interval of time, as new data arrives. Based on the presented cases, the performance of failure prediction systems can be characterized using many different complex metrics.

		Actual values		
		failure	*no failure*	
Prediction	*Failure predicted*	**TP**	**FP**	*Predicted* Positives, P'
	no failure predicted	**FN**	**TN**	*Predicted* Negatives, N'
		Positives P	Negatives N	

Figure 2.8 – The failure identification problem: the contingency table

Some examples are:

Precision

$$\frac{TP}{TP+FP} = \frac{Correctly\ predicted\ \boldsymbol{failures}}{All\ the\ predictions} \tag{2.8}$$

Recall / True Positive Rate / Sensitivity

$$\frac{TP}{TP+FN} = \frac{Correctly\ predicted\ \boldsymbol{failures}}{All\ the\ occurred\ failures} \tag{2.9}$$

Accuracy

$$\frac{TP+TN}{TP+TN+FP+FN} = \frac{All\ correctly\ predicted\ \boldsymbol{cases}}{All\ cases} \tag{2.10}$$

The prediction result 0/1 is obtained by applying a *threshold* to the output of the predictor, in general a numeric value: if the output is above (or below) the threshold, a failure (no failure) or 1 (0) is predicted. Different values lead to different behaviors of the model, also impacting the predictor performance (TP, FP, etc.). Adjusting the threshold results in different performance values, which may be a convenient way to tune a prediction model. For this reason, some methods were proposed to study the characteristic of a predictor when varying its prediction threshold.

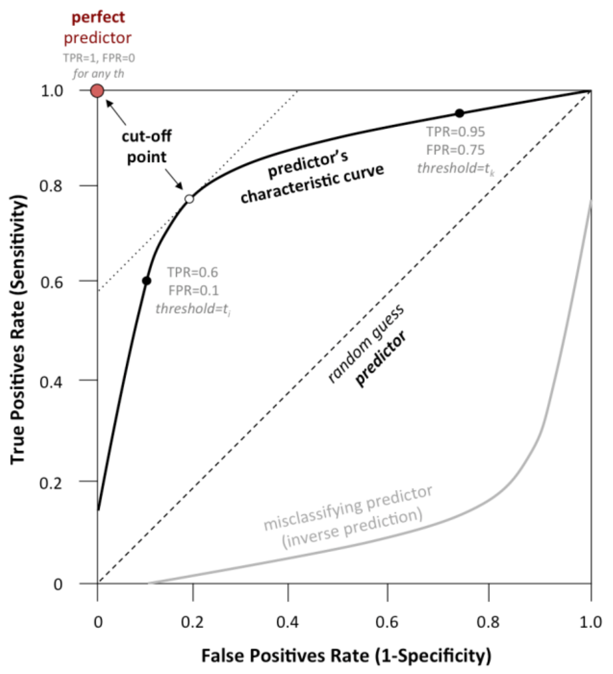
Receiver Operating Characteristic analysis is a widely accepted method for assessing the performance of binary classifiers, plotting the tradeoff between *True Positive Rate* (TP/(TP + FN), or *TPR* or *Sensitivity*) and *False Positive Rate* (FP/(TN + FP), or *FPR* or *1-Specificity*) (see Figure 2.9), as the threshold used for defining the predictions is given different values. The objective of an optimal training is to optimize the predictor in a way that maximizes the TPR and minimizes the FPR. Thus, the most the curve is near to the ideal classifier curve (i.e., the nearest it is to the upper-left corner, with AUC tending to 1), the best the classifier is able to predict or classify the target event, independently from the threshold value used. In this direction, a relevant performance indicator related to ROC analysis is the *ROC* *curve's cut-off point*, which is the point corresponding to the optimal performance values maximizing TPR and minimizing FPR.

A performance indicator index based on ROC analysis is the **ROC-AUC**, which is the measured area under the ROC curve). An example is presented in Figure 2.9 (a). The *ideal classifier* would give TPR=1 and FPR=0 for each threshold value (that is the best performance a classifier can obtain, as FP=FN=0), and its AUC is equal to 1. A *random guess classifier* has a nominal AUC=0.5, and its curve is usually taken as reference in the ROC space. If a classifier curve got AUC<0.5, usually one can invert the classified classes (Fawcett 2006). Thus, a ROC curve is identified by a non-continuous shape (due to the limited and discrete values of threshold used), and by an AUC usually greater than 0.5 (continuous shapes in Figure 2.9 are for presentation purposes only).

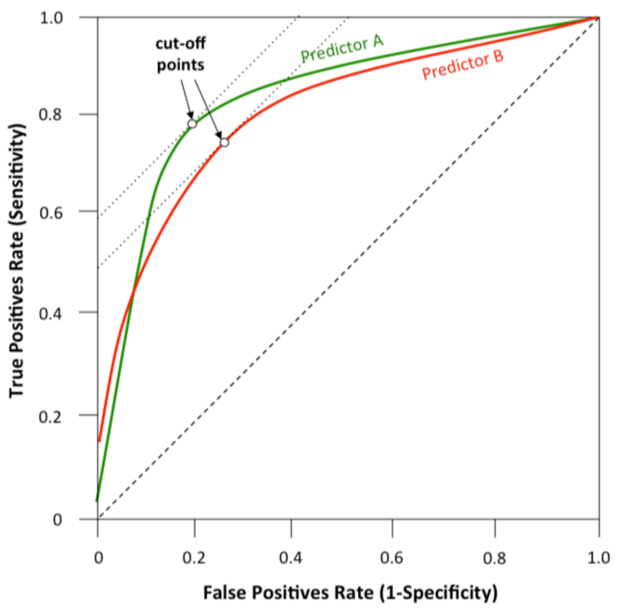
Figure 2.9 (b) presents the comparison between two ROC curves, where Predictor A is averagely better than Predictor B, despite its values in the left-bottom corner. In general, comparing predictors using ROC curves says much about their response when using different threshold values.

A similar performance assessment method is **Precision-Recall curve analysis** (Felix Salfner, Lenk, and Malek 2010; Davis and Goadrich 2006), obtained similarly to ROC curve, by computing the Precision and Recall of a model when varying the threshold used for the classification. Similarly to ROC curves, a perfect classifier would have a Precision-Recall curve reduced to the point (Precision=1, Recall=1), which corresponds to the upper-right corner of the plane. An example of Precision-Recall curve is shown in Figure 2.10, where the Predictor A is better than Predictor B.

A valuable advantage of the ROC analysis is that ROC curves are independent of the class priors ((Bradley 1997) (i.e., the distribution of the samples belonging to each class), as demonstrated in several works (e.g., (Chawla, Japkowicz, and Kotcz 2004; Chawla 2010; Zweig and Campbell 1993; Fawcett 2006; Wang 2008)). This makes ROC being a natural solution for evaluating binary classifiers in case of imbalanced datasets. In fact, when a dataset contains more positive than negative samples (or vice-versa), a classifier with fixed-threshold may present poor performance (Chawla, Japkowicz, and Kotcz 2004) that may improve when changing the threshold. The ROC analysis is independent from the decision of the threshold, thus giving a complete insight on the classifier performance, whatever threshold value is used. In practice, ROC analysis metrics have some significant characteristics, namely: independence from the dataset characteristics, capability for performing sensitivity analysis in the context of varying thresholds, easiness of interpretation of the results, and large usage for the assessment of information retrieval systems.



(a)



(b)

Figure 2.9 – A ROC curve

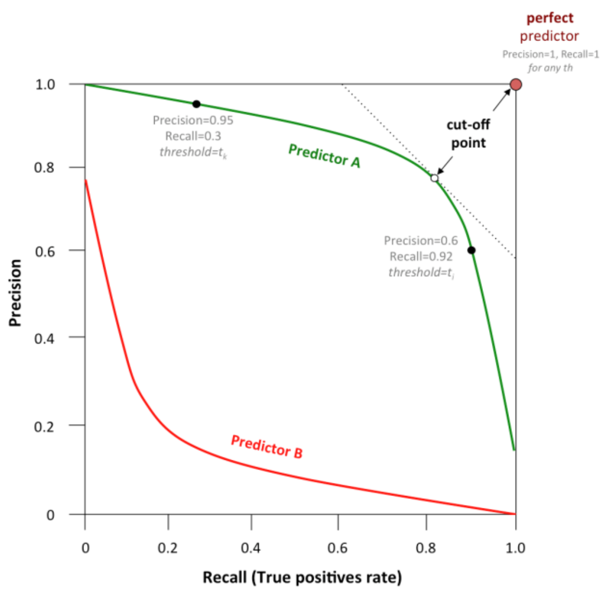


Figure 2.10 – A Precision-Recall curve

Comparing different failure prediction systems is not trivial, even if good metrics are used. In fact, the assessment of the prediction performance is only possible if the prediction systems share the same concepts, as for instance the same model for the prediction problem.

The problem of **comparability** of online failure prediction approaches was also highlighted by Salfner et al. in their survey (F. Salfner, Lenk, and Malek 2010). In fact, Salfner et al. (F. Salfner, Lenk, and Malek 2010) were the first to define comparability of failure prediction approaches as a property that "can only be achieved if two conditions are met: (i) a set of standard *quality evaluation metrics* is available, and (ii) *publicly available reference data sets* can be accessed". However, only few metrics for evaluation are proposed in their work. On the other hand, although there are some initiatives for building repositories for failure datasets, as the Computer Failure Data Repository (Usenix and Carnegie Mellon University (CMU) 2006) that publicly provides detailed failure data from several systems, such repositories are not enough for assessing and comparing failure prediction algorithms meant to be used on a particular system. In fact, to understand the effective performance of a failure prediction algorithm, it has to be tested in the system where it will be used using a rigorous experimental process.

2.2.5 Building and optimizing Online Failure Prediction models

The failure prediction literature includes prediction models built *explicitly* (manually) by describing its characteristics, using an informal or formal language, as mathematics, and *implicitly* by using training algorithms that associate the failure event – after it happened – with information about events occurred before the failure occurred. The automatic training procedure is based on the use of datasets built from the data collected during system execution (when a failure or no failure occurs, depending on the model to be built), correlated with the observed failures (*failure coding*) and organized in training datasets (*TDSs*) and testing datasets (*TTDSs*), the latter used to assess the prediction performance after training. The division in training and testing sets is done according to a certain percentage (e.g., half of the data samples are used for training, and the other half is used for testing), depending on the problem. The association of failure events with data can be performed manually, or by using failure detectors, i.e., models that can automatically detect the occurrence of a failure. Similarly to the failure prediction models, **failure detection** models can be built manually (e.g., by defining rules identifying a failure event) or using learning algorithms.

Failure coding is the association of a failure event to a set of failure-related data, which is a step required when training failure prediction algorithms. Coding approaches depend on the user needs and on the type of prediction technique used. For instance, when using classifiers, the failure data must be divided in samples and each sample should be labeled with a 0 (i.e., no-failure) or a 1 (i.e., failure). Moreover, the data associated to the failure must be relative to a specific past time interval ending at the failure time. The failure prediction problem model proposed by the authors of the survey (Felix Salfner, Lenk, and Malek 2010) define such interval as the failure data window Δt_d.

Building failure predictors is an optimization procedure, as models are characterized by several parameters, whose values impact on their performance. Each predictor should hence be assessed using a set of values that maximizes its performance, and several techniques can be used in this optimization process. In this scenario, one of the most critical points in building a prediction model is the choice of the system variables (or parameters) to monitor, representing the current state of the system. A wrong set of variables can make the prediction of a failure useless, as pointed out by Hoffman et al. in (G.A. Hoffmann, Trivedi, and Malek 2007), while the choice of a good set of variables can be even more important than the model (linear or non-linear (G.A. Hoffmann, Trivedi, and Malek 2007)) for obtaining an optimal prediction of a failure. To fully characterize a system, one often needs hundreds or even thousands of features. However, a huge set of features increases the complexity of the model that uses these features (i.e., that represents the system), being often too much complex to be used in reality. In this scenario, there is a need for a systematic approach to choose the variables for prediction.

Feature selection is a process that allows the systematic selection of a subset of features as an optimization problem (H. Liu and Yu 2005), being a crucial task for reducing the complexity of the models. The features represent the dimensionality of a domain where an object, system, and others, can be represented. It has been demonstrated that the problem of finding the optimal feature subset can be NP-complete or even NP-hard problems (Blum and Rivest 1992; Guyon and Elisseeff 2003). A typical feature selection process consists into several steps, namely:

1) **Subset generation**. A search procedure that produces a set of candidates, to be evaluated in the next step. A search strategy is needed.
2) **Subset evaluation**. Each subset is evaluated according to some evaluation criterion, and compared with the previous subsets. Usually, if the newest is better that the last one, it becomes the best candidate.
3) **Stopping criterion**. The process is repeated several times, until some stopping criterion is reached. At this point, the search is finished and the remaining subset is considered the optimal (or the best one).
4) **Result validation**. The subset found at the end of the search is validated against prior knowledge or different test sets.

There are several methods for solving the problem of variable (or feature) selection, which can be divided into two main groups (John et al. 1994): the **filter** and the **wrapper** approaches. In the former (*filtering*) the feature selector filters the irrelevant attributes independently of any specific learning algorithm. In the latter (*wrapping*), the most important features are filtered taking into account the specificities of the underlying learning mechanism. In addition to these two, a *hybrid* approach tries to take the advantages of both filtering and wrapping and uses both a relevance measure for the chosen set of variables that is independent from the learning algorithm, and a mining algorithm to find the best sub-set.

In addition to this categorization, Liu and Yu (H. Liu and Yu 2005) proposed a more complete categorizing framework for feature selection algorithms in 2005. The existing approaches are grouped according to three dimensions:

- **Evaluation criteria**, thus dividing the approaches in *Hybrid*, *Filter*, and *Wrapper*;
- **Search strategies**, considering a division according to the search strategy used, namely *Complete*, *Sequential*, and *Random*;
- **Data mining tasks**, as the availability of class information affects the evaluation criteria used by the feature selection algorithm. Thus, they consider *Classification* and *Clustering* for distinguishing the two cases.

It is worth noting that the classification proposed by Liu and Yu (H. Liu and Yu 2005) considers only the feature selection approaches used for classification and clustering, not including other tasks as association rules, regression, etc.

Hoffmann et al. (Günther A. Hoffmann 2004) proposed in 2004 an approach called Probabilistic Wrapper Approach (PWA), initially defined by Liu and Setonio (Huan Liu and Setiono 1997) in 1997. PWA is a hybrid feature selection method that tries to conciliate the best of filtering and wrapping by playing with combination of variables. However, to the best of our knowledge, these are the only approaches for selecting features for an optimal failure prediction that can be found in literature. In practice, the most relevant works available in literature show that the variable selection is mostly done manually and the variables are typically chosen on the behalf of the experience of the developer coming from previous works. However, Hoffmann et al. in (G.A. Hoffmann, Trivedi, and Malek 2006) demonstrated that variable selection is necessary for having an optimal model, and also that the choice of the variables is much more important than the choice between using a linear or non-linear model.

2.3 Virtualization and Online Failure Prediction

In the last decade the concept of **Virtual Machines** became very popular due to the possibility to simulate (one or more) real machines on one single real hardware machine. Both end-users and companies were charmed by the potential of this technology, which would permit to run several OS on just one machine, thus extending the usability of some software products, reducing the costs associated to a single machine when deploying complex systems, and having the possibility to increase some characteristics as security and dependability. Virtualization also gives the abstraction of having a different machine from the one whose hardware is actually in use, by providing a software layer implementing the low-level functions of the hardware. The first virtualized system was the PR/SM Hypervisor from IBM Corp., which was used to share a single 370 Mainframe system among several users (one machine, many users) (Rose 2004).

With the growt in performance of hardware systems, virtualization started to be used more and more in order to have a single hardware machine hosting several Virtual Machines. Virtual Machines are usually managed by a VMM (Virtual Machine Monitor), or Hypervisor, which takes care of forwarding the virtual-hardware requests to the “real” hardware and, in case of several virtual machines in the same physical machine, also manages the multiplexing of the existing hardware to the virtual OSs. Virtualization technologies can be classified in two families (Figure 2.11) regarding the type of Hypervisor implemented:

- **Type-I Hypervisors**, directly interfacing with the physical machine, and managing the sharing of resources by directly accessing them (usually, a standalone server).
- **Type-II Hypervisors,** hosted by an existing OS (is the case of the so called "desktop virtualization" systems). In this case the Hypervisor is a software layer between the host OS and the guest VM, and manages the machine hardware through the host OS's drivers.

While in the second case the Hypervisor is limited by the operations that the hosting OS provides, in the first case the management of the Virtual Machines is made easy by the direct access to the physical machine. Type-II Hypervisors spread in the last years thanks to specific advances on hardware towards virtualization and their performance, and nowadays are widely used especially for server consolidation (Khanna et al. 2006).

Virtualization technology offers several other features, as live migration of the VMs, performance isolation, and security mechanisms. For this reason, the use of virtualization solutions in secure and dependable systems is nowadays not rare. In this scenario, failure prediction has been pointed out as a solution for helping in implementing large, virtualization-based, dependable systems. For example, Polze et al. (Polze, Troger, and Salfner 2011) proposed an architecture for high-availability and high-performance systems based on virtualization, where the use of live migration is triggered by online failure prediction, using indicators on the health of the system. Other similar work is (Nagarajan et al. 2007) that automatically migrates processes from "unhealthy" nodes to healthy ones in a Xen environment. Fu et al. (Fu 2009) proposes a reconfigurable distributed virtual machine (RDVM) infrastructure with failure-aware node selection to be used for high-availability computing. Reiser et al. (Reiser and Kapitza 2007) uses an hypervisor to initialize a new replica in parallel to normal system execution, focusing on minimizing the proactive migration time, which can interfere with system operation.

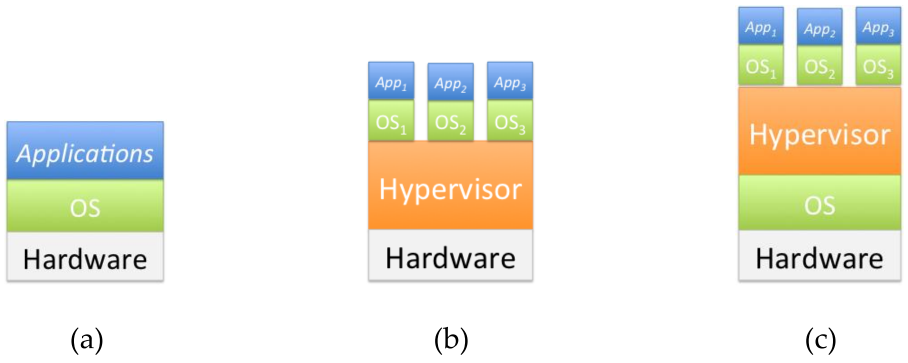


Figure 2.11 – No virtualization (a), Type-I Hypervisor (b), Type-II Hypervisor (c)

2.4 Fault Injection

Fault injection is an experiment-based approach that deliberately introduces faults into a computer system in a way that emulates real faults (Arlat, Crouzet, and Laprie 1989), with the goal of observing its behavior. The deliberate injection of faults can help understanding the impact that residual faults (including hardware and software faults) have on a system. Fault injection has been used in many works where the observation of systems in the presence of faults is important, such as fault tolerance and dependability validation (Arlat et al. 1990; J. Duraes and Madeira 2003), estimation of fault-tolerance parameters (Arlat, Crouzet, and Laprie 1989), and dependability benchmarking (J. Duraes, Vieira, and Madeira 2004).

Injecting faults means to mimic the presence of a hardware or software fault: **hardware faults**, such as bit-flip and stuck-at, occurring in hardware components, and **software faults**, representing defects that remained in a piece of software due to some issue during the development phase. Faults can be emulated by hardware- or software-based techniques (e.g., (Hsueh, Tsai, and Iyer 1997)), though software faults are more likely to be emulated by software techniques only. The implementation of a fault injection tool depends on the target system to be analyzed, on the faults to be injected, on the access to the injection locations, just to name a few.

Hudak et al. (Hudak et al. 1993) is among the first works on fault injection: the authors compared techniques as n-version programming, recovery blocks, concurrent error-detection, and algorithm-based fault tolerance using both hardware faults (e.g., code and data corruption) and software faults (simulated design-faults including control flow, array boundary, computational, and post/pre increment/decrement software mutations). In a more general sense, fault injection has been used to assess dependability properties of computer systems, as for example in the works from Koopman and Madeira (Koopman and Madeira 1999) and Vieira and Madeira (M. Vieira and Madeira 2003). Following the injection of hardware faults that emulate the effects of physical defects and external causes, during the last two decades fault injection started to focus on software faults due to the increasing complexity of software when compared to hardware components. The first techniques able to emulate the effects of software-faults were developed by Christmansson and Chillarege (Christmansson and Chillarege 1996) in 1996, Koopman et al. (Koopman et al. 1997) in 1997, and Fabre et al. (Arlat, Fabre, and Rodriguez 2002) in 1999.

Fault injection can be used to directly assess the impact of specific errors in the system, thus allowing collecting information that can be used for improvement. For example, Koopman et al. in (Koopman et al. 1997) injected software faults in the OS API for testing the robustness of five operating systems: Mach, HP-UX, QNX, LynxOS, and Stratus FTX. Among the existing software fault injection techniques, G-SWFIT (J. A. Duraes and Madeira 2006) appears as a reference. In fact, the technique developed at the University of Coimbra is the *de-facto* standard in the emulation of generic software faults (i.e., faults can be found in generic software systems), while addressing the *fault representativeness* problem (i.e., the property of an injected fault to exist in real software systems).

2.4.1 Fault injection environment and a taxonomy

A fault injection is usually made of basic components and organized in a **fault injection environment** (Hsueh, Tsai, and Iyer 1997), consisting of two main components: the **fault injection tool**, and the **target system** (i.e., the system on which the injection is performed). The fault injection tool injects faults into the system, at runtime or when the system is offline. A representation of the components that usually compose a fault injection environment can be found in (Hsueh, Tsai, and Iyer 1997), and are presented in Figure 2.12, namely: a *controller* (that controls the fault injection experiment), a *fault injector* (that introduces faults and must be the less intrusive possible), a *fault library* (that specifies which faults to inject, where, and when), and a *monitoring system* (for catching the effects of the fault on the system, usually working together with a data collector and analyzer). A *workload generator* is often needed also to exercise the system.

There are two main types of fault injection approaches: **hardware fault injection** and **software fault injection**. The former can reproduce or emulate the effects of hardware faults (e.g., a bit-flip caused by high levels of radiations), and may be implemented using hardware tools (i.e., tools that include a big portion of specific hardware for the injection of the faults) or software mechanisms (typically named as SWIFI – Software Implemented Fault Injection). On the other hand, software faults are emulated by software approaches only, although some studies showed that some software faults could be emulated by injecting hardware faults (Madeira, Costa, and Vieira 2000).

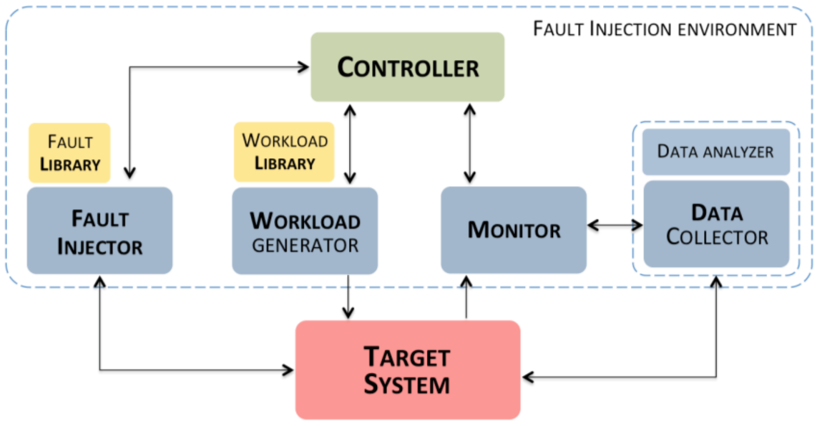


Figure 2.12 – A fault injection environment

2.4.2 Injection of hardware faults

Hardware fault injection consists of an operator able to introduce faults in the target system through a tool, which can be a physical tool, a software tool, or both. Fault types that can be inserted in the target system are presented in Table 2.1, representing the erroneous situations in which hardware parts may incur. For instance, open or short circuits may occur due to environmental conditions (e.g., dust, liquids, humidity), bit-flips are temporary changes in the state of one or more flip-flops due to external causes (e.g., strong radiation or electro-magnetic field), and stuck-at are caused by hardware defects or aging.

Physical hardware fault injection approaches can be divided in two main categories (Hsueh, Tsai, and Iyer 1997): *with contact* and *without contact*. The first category includes fault injection systems where the injection tool is physically in contact with the target system. This kind of injector is often called *pin-level injector*, as the injector has its direct contact with the pins of the circuit. The two main techniques used in this context are *active probes* (i.e., probes attached to the pins that send electric signals to the circuit pins) and *socket insertion* (a socket inserted between the target system and the board that allows to inject more complex logic faults). An example of *pin-level injection* is the MESSALINE tool (Arlat, Crouzet, and Laprie 1989), developed at LAAS-CNRS, in Tolouse, France. The injector uses active probes to alter the voltage applied to the pins (reproducing basically stuck-at faults) and also the socket insertion technique. The injection system was used to validate a distributed communication system for transportation systems, within the Esprit Delta-4 Project. Messaline could inject stuck-at, open, bridging, and complex logical faults.

The second category of hardware fault injectors (without contact) includes systems that emulate the presence of faults without any contact with the board of the target system. Such systems use, for instance, the generation of electromagnetic fields and, most frequently, heavy-ion radiations on the components of the system. The FIST (Fault Injection System for Study of Transient Fault Effect) tool, developed at the Chalmers University of Technology, Sweden, uses both contact and contactless methods (Gunneflo, Karlsson, and Torin 1989), using heavy-ion radiations that lead to transient faults in random locations inside the exposed chip. The EMI (Electro-Magnetic Interference) tool (Karlsson et al. 1998), on the other hand, uses only the

Table 2.1 - Hardware faults model

Faults	Description
Open	Always “open” line
Bridging	Short-circuit
Bit-flip	Inverting a bit (0-1 or 1-0)
Spurious current	Bit randomly left at 0 or 1
Power surge	A transient disturb in the power supplied to the hardware
Stuck-at	Bit always at the same value (0 or 1)

contactless method, being the electromagnetic fields generated using two charged plates, which cause faults in the target system placed in the middle of the two.

The injection of hardware faults moved then to a new generation of injection tools that are known as **SWIFI** (Software-Implemented Fault Injection). The injection of faults by software is carried out by inserting errors in the system structures that can be directly accessed through software (e.g., memory, processor registers, some peripheral devices, etc.) or that can be accessed by software in an indirect way including many internal processor structures (e.g., cache, integer unit, floating point unit, decoding unit, etc.). SWIFI approach has become very popular due to its low complexity and low development effort required, when compared to fault injection based on hardware level. Some examples of SWIFI fault injection tools are Ferrari (Kanawati, Kanawati, and Abraham 1992), FTAPE (Tsai and Iyer 1995), and Xception (Carreira, Madeira, and Silva 1998). As an example, the Ferrari (Fault and ERRor Automatic Real-time Injection) tool, developed at the University of Austin (Texas) (Kanawati, Kanawati, and Abraham 1992), uses software traps and trap handling mechanism to inject CPU, memory, and bus faults. The tool includes four components: the initializer and activator, the user information, the fault and error injector, and the data collector and analyzer. A fault (e.g., a modification of the Program Counter value that emulates a bit-flip) is coded in a trap handling routine, and injected when the trap is caught and the routine is executed. The faults injected by can be transient or permanent, and among the ones emulated we can find address line errors, data line errors, and condition bit errors. On the other hand, Xception injects faults in registers of the processor by taking advantage of debugging and performance monitoring features present in modern processors. In practice, the tool executes small exception routines that implement the fault injection by modifying the interrupt handler vector. Such technique has shown to be particularly useful for evaluating the robustness of user applications and operating systems, requiring no modification of the application software and no insertion of software traps.

2.4.3 Injection of software faults

Software fault injection consists of emulating residual faults that remain in software after the testing process at different development levels. The issue of injecting faults that emulate software defects was addressed relatively late (the first work was published in 1996 (Christmansson and Chillarege 1996)), also due to the high complexity of software faults and of their emulation. The injection of software faults is often called **software fault injection**, although in some works the expression “software fault injection” is still used to name SWIFI approaches (presented above). In this work we focus on the injection of software faults, as these are nowadays the largest cause of failures in computer systems (Lee and Iyer 1995; Kalyanakrishnam, Kalbarczyk, and Iyer 1999).

The emulation of software faults raises several difficulties, such as **what**, **where** and **when** to inject. The key problem is the definition of a **fault model**, i.e., what types of faults to inject. Some works proposed approaches based on the analysis of faults present in several software products (e.g., (Christmansso and Rimén 1998; Christmansson and Chillarege 1996)). The first model for software faults is the **Orthogonal Defect Classification** (ODC) by Chillarege et al. (R. Chillarege 1995), which encompasses software defects and triggers. The idea was to provide insights about the quality of the development process of software systems, focusing on knowing the percentage of a certain type of faults affecting the system, their cost, the cost for their correction, as well as the distribution of team-force in the various phases of the system development. In practice, the ODC classifies software faults according to the way a programmer can correct them:

- **Assignment**: value(s) assigned incorrectly or not assigned at all.
- **Checking**: missing or incorrect validation of data or incorrect loop or conditional statements.
- **Interface**: errors in the interaction among components, modules, device drivers, call statements, or parameter lists.
- **Timing/serialization**: missing or incorrect serialization of shared resources.
- **Algorithm**: includes efficiency or correctness problems that affect a task and can be fixed by (re)implementing an algorithm or data structure without the need of a design change.
- **Function**: a defect that significantly affects capability, end-user features, API interface, interface with hardware architecture, or global structure. A certain amount of code is either implemented incorrectly or not implemented at all, thus needs a formal design change.
- **Build/package/merge**: errors due to mistakes in library systems, management of changes, or versions control.
- **Documentation**: errors that can affect both development documentation and maintenance notes.

Madeira et al. (Madeira, Costa, and Vieira 2000) were the first to use ODC for defining the faults that could be injected by the Xception fault injection tool (Carreira, Madeira, and Silva 1998). In 2006, Durães et al. (J. A. Duraes and Madeira 2006) pointed out the inadequacy of using the ODC in the context of software fault injection. Although ODC has been successfully used to improve the software designing process and provides an important basis for understanding and classifying software faults, it relates faults to the way they are corrected (which can be done in different ways) and not with the way they can be emulated.

A fault classification that extends ODC was proposed by (J. A. Duraes and Madeira 2006) based on a field study. The main idea is that a defect is one or more programming language constructs (statements, expressions, function calls, etc.) that are either missing, wrong, or in excess. The authors analyzed the evolution of several open-source software applications (analyzing the bugs and how they were corrected), and classified each fault according to its nature, which can be one of the following: missing construct, wrong construct, or extraneous construct (i.e., a construct that is superfluous). In practice, Durães and Madeira proposed an orthogonal extension to the ODC classification where a fault belonging to one of the Assignment, Checking, Interface, Timing/Serialization, Algorithm and Function classes can be relative to a missing, wrong or extraneous construct. Table 2.2 presents, for each dimension, the faults most frequently observed in the software applications studied, and the classification of each fault type presented according to ODC model. A major observation by Durães and Madeira (J. A. Duraes and Madeira 2006) is that more than 50% of the software faults can be "realistically" emulated by a small set of generic fault types, which correspond to the most frequent fault types found in real software.

The injection of a specific type of software fault can be performed only in specific parts of the system code (**where**), namely parts of the code that can be changed to emulate a fault. For instance, emulating a "missing conditions of an *if* statement" (see Table 2.2) is possible only modifying the *if* statements present in the code. The parts of the code where a fault type can be emulated are referred to as *fault locations*.

The insertion of a fault in a software component (i.e., **when** it is injected) can be done *offline* (at *compile-time* by modifying the source code) or *post-deployment* (by injecting directly in the binary code), at a specific instant of time. It is worth noting that although offline injection is easier to implement, source code is often not available.

Table 2.2 - Fault coverage of fault types from (J. A. Duraes and Madeira 2006)

Fault nature	Fault specific types	# Faults	ODC types				
			ASG	CHK	INT	ALG	FUN
Missing	*if* construct plus statements (**MIFS**)	71				✓	
	AND sub-expr in expression used as branch condition (**MLAC**)	47		✓			
	function call (**MFC**)	46				✓	
	if construct around statements (**MIA**)	34		✓			
	OR sub-expr in expression used as branch condition (**MLOC**)	32		✓			
	small and localized part of the algorithm (**MLPA**)	23				✓	
	Variable assignment using an expression (**MVAE**)	21	✓				
	functionality (**MFCT**)	21					✓
	variable assignment using a value (**MVAV**)	20	✓				
	if construct plus statements plus *else* before statements (**MIEB**)	18				✓	
	variable initialization (**MVIV**)	15	✓				
Wrong	logical expression used as branch condition (**WLEC**)	22		✓			
	algorithm – large modification (**WALL**)	20					✓
	value assigned to variable (**WVAV**)	16	✓				
	arithmetic expression in parameter or function call (**WAEP**)	14			✓		
	data types or conversion used (**WSUT**)	12	✓				
	variable used in parameter of function call (**WPFV**)	11			✓		
Extraneous	variable assignment using another variable (**EVAV**)	9	✓				
Total faults for these types in each ODC type		452	93	135	25	192	41
Coverage relative to each ODC type (%)		68	65	81	51	72	100

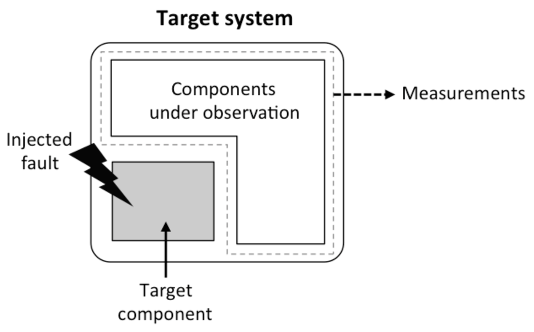


Figure 2.13 – Software Fault Injection and system observation (J. A. Duraes and Madeira 2006)

A fundamental aspect in the fault injection scenario is the clear separation between the target component and the part of the system under observation. In fact, the emulation of software faults always requires the introduction of small changes in the target code, and any conclusions about the faulty-component may be misleading, as the injected component is different from the original one. In practice, the goal is generally to evaluate how the rest of the system copes with such faulty component (see Figure 2.13), considering the target component as faulty. This is quite natural for software faults as a component-based approach is generally used for architecting software.

2.4.4 Generic Software Fault Injection Technique (G-SWFIT)

Among the several software fault injection tools available in literature, Durães and Madeira propose the most complete novel approach for injecting software faults when the source-code is not available, thus being very relevant for software using OTS (Off-The-Shelf, i.e., third party) modules and OTS-based systems (J. A. Duraes and Madeira 2006). The approach, named Generic Software Fault Injection Technique (G-SWFIT), is based on the use of educated mutations at machine-code level that emulate software faults at high-level coding. The G-SWFIT is based on a **fault library** that includes emulation operators for each fault type, based on *machine-code level patterns* (which identify the constructs that can host the specific fault) and the corresponding *code changes* (representing the translation of a single software fault into machine-code).

The types of software faults in the fault library were obtained from a field study of the most occurring faults in software systems: the authors analyzed several open-source software systems and classified the faults corrected along their development using an extension of the ODC classification (as mentioned above). This resulted in a set of the emulation operators relative to the most frequent faults. The ones included in the G-SWFIT technique are presented in Table 2.3.

The injection of software faults is done only where a fault is likely to exist (thus being *context-based*), through the automatic search of the machine-code level patterns defined in the *library* for each software fault type. The authors validated the injection technique by comparing the translation of the software faults injected at high-level in the source-code of three software applications (among which GZip) with the low-level patterns defined by the compiler. In most cases, their tool reached the maximum accuracy in emulating software faults. As the fault locations are identified before the actual injection and the set of faults is generated based on this information, the faults can be injected offline or online, with low intrusiveness. However, it is worth noticing that the injection at runtime can present representativeness problems, as the fault location could have already been executed before the fault is injected.

Figure 2.14 shows the basic functional schema of the G-SWFI Technique. The system code is disassembled in order to translate the executable file into assembly code that can be scanned for target code patterns. Afterwards, the tool reads and generates a mutant with a fault injected. The mutated versions of the single slice of code are then assembled to generate mutated executable versions. Alternatively, during online injection, the mutations can be injected directly in the associated process in memory. It is worth noting that online injection strongly depends on the possibility to access

Table 2.3 - Most frequent fault types found in (J. A. Duraes and Madeira 2006)

Fault types	Description	% of total observed	ODC classes
MIFS	Missing "If (cond) { statement(s) }"	9.96 %	Algorithm
MFC	Missing function call	8.64 %	Algorithm
MLAC	Missing "AND EXPR" in expression used as branch condition	7.89 %	Checking
MIA	Missing "if (cond)" surrounding statement(s)	4.32 %	Checking
MLPC	Missing small and localized part of the algorithm	3.19 %	Algorithm
MVAE	Missing variable assignment using an expression	3.00 %	Assignment
WLEC	Wrong logical expression used as branch condition	3.00 %	Checking
WVAV	Wrong value assigned to a value	2.44 %	Assignment
MVI	Missing variable initialization	2.25 %	Assignment
MVAV	Missing variable assignment using a value	2.25 %	Assignment
WAEP	Wrong arithmetic expression used in function call parameter	2.25 %	Interface
WPFV	Wrong variable used in parameter of function call	1.50 %	Interface
	Total faults coverage	**50.69 %**	

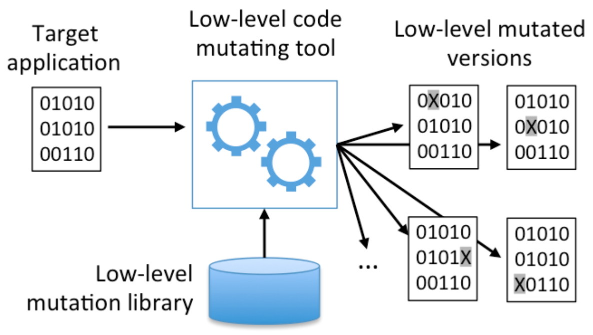


Figure 2.14 – The G-SWFIT injection task

the process in memory or the processor's registers. The software fault injection process may result in a large number of mutants that may make experiments infeasible in a short time.

There are several implementations of the G-SWFI technique nowadays, as for instance for Java environments (the J-SWFIT tool from Sanches et al. (Sanches, Basso, and Moraes 2011)), and the tools by Durães for C/C++ environments (J. A. Duraes and Madeira 2006). The adoption of the G-SWFIT recommendations can be seen in Barbosa (Barbosa et al. 2007), Basso et al. (Basso et al. 2009), Fonseca et al. (Fonseca, Vieira, and Madeira 2007) , Moraes et al. (Moraes et al. 2007), Natella et al. (R. Natella and Cotroneo 2010; R. Natella et al. 2010; Cotroneo, Fucci, and Natella 2012), among others.

The ultimate question that arises from the use of software fault injection, including the G-SWFIT technique, is to demonstrate that the injected faults are representative of a situation in which the system contain faults that escaped the several testing phases (from code inspection to functional testing), which has no trivial answer.

2.4.4.2 The problem of representativeness in software fault injection

A software defect permanently lies in the code of system. Besides the fact that its behavior is often soft *(transient)*, some software faults elude the testing phase due to insufficient testing effort, but also due to the complexity of its activation, which makes it "hiding" from testing. The injection of software faults must aim at emulating such types of faults for mimicking realistic scenarios, which is obviously not possible because these defects are not known in advance (if we knew the bugs, we would fix them beforehand). Given this impossibility, the correct emulation of software faults by fault injection requires (J. A. Duraes and Madeira 2006):

- **The identification and characterization of the most important classes of software faults and estimation of the relative percentages of these classes in real programs**. The relevant faults are those that correspond to representative fault types of the real residual defects found in deployed software systems (J. A. Duraes and Madeira 2006).

- **Techniques to inject faults that generate errors or induce erroneous program behavior similar to the ones caused by specific classes of real software faults**. In other words, what is important is to avoid the injection of faults that cause errors that would not be generated by a real software fault (J. A. Duraes and Madeira 2006).
- **Considering the elusive nature of a software fault to emulate**. In fact, the faults that are likely to be present in the system operational phase are faults that escaped the testing phases. Natella and Durães (R. Natella et al. 2010) demonstrated that a necessary condition for the representativeness of emulated software faults is their elusive nature.

Regarding the elusive nature of the faults, a first model was proposed by Jim Gray (Gray 1986) (Bohrbugs and Heisenbugs), and successively integrated by Grottke and Trivedi (Grottke and Trivedi 2005). A fault can be classified as a **Bohrbug**, causing a failure under simple and known conditions (being hard or non-elusive) or a **Mandelbug**, whose manifestation is non-deterministic, causing failures under complex and unknown conditions (being a soft or elusive software fault). In addition, to these two types, **Heisenbugs** are faults that stop "causing a failure or that manifests differently when one attempts to probe or isolate it" (Grottke and Trivedi 2005), in a similar way to what happens in physics with waves and particles (i.e., electrons), according to the Heisenberg principle. Heisenbugs are considered a sub-type of Mandelbug, and examples of their manifestation are when some *debuggers initialize unused memory* to default values, thus eliminating failures due to improper initialization, or the *influence on process scheduling* when trying to investigate a failure. Grottke and Trivedi also define software-aging bugs as Mandelbugs and in some cases as Heisenbugs (Grottke and Trivedi 2005).

Fault injection should emulate Mandelbugs, or Bohrbugs supposing that the system was poorly tested. Natella and Durães (R. Natella et al. 2010) analyzed the *elusiveness* of a set of faults injected using the G-SWFIT technique into a MySQL server, finding that almost 85% of injected faults eluded 50% of the test cases defined. The remaining 14.57% of non-elusive faults should be eliminated *a priori* (i.e., without knowing the system test cases). The authors proposed a way in which the representativeness of the injected faults is improved, by means of a set of criteria based on code metrics for *excluding non-elusive faults* from the faults to be injected.

The concept of **software fault trigger**, introduced for the first time by Sullivan and Chillarege in (Sullivan and Chillarege 1991), also helps in modeling the activation of a fault that is dormant in a software system. In practice, modeling and understanding the nature of a software fault in terms of triggering conditions is useful for understanding which are the faults that mostly hide during the testing phase, and that are more likely to activate when not expected. Chillarege et al. (R. Chillarege 1995) defined several dimensions for software faults triggers, including workload volume/stress, system start-up and restart, hardware and software configuration, and normal mode.

2.5 Fault Injection and Online Failure Prediction

Failures are rare events, and the collection of training data for prediction systems in a short time frame can take long time (G. Hoffmann and Malek 2006)(Li, Vaidyanathan, and Trivedi 2002). Furthermore, even if one is able to collect failure data, those data (collected in a specific time period) may not be representative of the system behavior in other periods, due to runtime systems evolution (e.g., workload variation, software upgrades). In this work we argue that a potential solution to this problem is the deliberate injection of realistic faults.

The literature presents few works related to the failure prediction domain in which the authors try to accelerate the experiments injecting software faults. Gross et al. (Gross, Bhardwaj, and Bickford 2002) injected memory leaks to have controllable parasitic resource consumption rates, to speed up the experiments and fine-tuning the MSET (Multivariate State Estimation Technique). In (Alonso et al. 2010) and (Alonso, Torres, and Gavaldà 2009) the authors also used the injection of memory leaks to accelerate their experiments and to demonstrate the effectiveness of the M5P algorithm (a Decision Tree algorithm) for predicting aging-related failures occurring in a Tomcat web server.

Even though some works already used the “injection” of software bugs (mainly memory leaks) to accelerate and validate the proposed prediction models, some limitations may be highlighted, taking into consideration the concept of software fault injection presented in the previous section. For example, in the work of (Alonso et al. 2010; Alonso, Torres, and Gavaldà 2009), it is not clear if the authors injected memory leaks at runtime and which kind of methodology they used for doing it. There is also no assumption on the distribution of the faults, consequently raising questions regarding the representativeness of the reproduced failure model (and modes). Moreover, most works target only aging-related failures and do not cover the entire spectrum of the possible failures in which a computer system can incur.

2.6 Computer systems benchmarking

A **benchmark** is an instrument that allows evaluating and comparing different entities (systems, components, tools, etc.) according to specific characteristics, like for example performance, robustness and dependability, under the same conditions (Gray 1993). In practice, benchmarking is a process that encompasses the execution of the system under test under conditions that are constant over time and the measurement of specific characteristics at each execution, in a way that provides results that are fair and comparable across alternative systems and/or components.

The main components of a benchmark are:

- **Metrics** that characterize the objects under comparison. For instance, metrics for benchmarking CPUs' throughput are *Instructions Per Second* (typically scaled to millions, MIPS, or above – GIPS, TIPS) and *Floating-Point Operations Per Second* (e.g., MFLOPS), while complex computer systems as web servers are analyzed with respect to their *response time*, *availability* and *latency* ("Transaction Processing Performance Council (TPC)"). The definition of metrics is of utmost importance for modeling the characteristics of the system to be measured in a proper way;
- **Workload**, which is a set of operations that the systems under test must execute during the benchmark execution, usually including several *components* (instructions, software components, other systems) and *parameters* (defining a particular instance of the workload). Workloads are typically built according to the characteristics of the system under benchmarking. For instance, workloads for measuring the CPU throughput in terms of FLOPS must be made of floating-point, computation-intensive instructions, while measuring the response time of a web server requires a set of several remote nodes requesting operations that the system must execute at a given rate. Several techniques are available for defining a proper workload (Calzarossa, Italiani, and Serazzi 1986; Agrawala, Mohr, and Bryant 1976; Calzarossa and Serazzi 1993; D. Ferrari 1972; Eeckhout et al. 2005; Domenico Ferrari 1984), which nonetheless remains an open problem in many scenarios;
- **Benchmarking procedure** that describes the setup required to run the benchmark and the set of steps and rules to be followed during its execution (Gray 1993). For instance, benchmarking a web server requires setting-up of the remote nodes submitting the workload, configuring the environment to automatically start the web server, starting and stopping the workload execution, calculating the defined metric, among others.

In order to give confidence on the results, a proper benchmark must encompass several **properties** (M. Vieira and Madeira 2003), namely it should be easy to implement and use, provide repeatable results, be portable to different systems in a given domain, include representative components, and be non-intrusive in order to not interfere in the results.

Work on performance benchmarking ranges from simple benchmarks that target a very specific hardware system or component to very complex benchmarks focusing on complex systems (e.g., databases, operating systems, web servers (M. Vieira and Madeira 2003)). Performance benchmarks have contributed to improve successive generations of systems (Gray 1993), and the beginning of the millennium has boosted the research on dependability benchmarking, with several works carried out by different groups following different approaches (e.g., experimentation, modeling, fault injection) (Koopman et al. 1997; M. Vieira and Madeira 2003; Zheng 1993; Antunes and Vieira 2010).

The goal of dependability benchmarking is to characterize the behavior of a system in the presence of faults, quantifying dependability attributes. A dependability benchmark thus involves the use of techniques as fault injection and robustness testing, adds to the main components of a benchmark a *faultload* (containing the faults in presence of which assess the system), and measures relative to *dependability attributes*.

In the last few years, benchmarks were also developed for evaluating the security of systems, as for example (Mendes, Madeira, and Duraes 2014; Marco Vieira and Madeira 2005; Mendes, Duraes, and Madeira 2011). Such benchmarks are based on the idea of evaluating a system in the presence of vulnerabilities related to its security (i.e., software faults that have the effect to reduce the security attributes of a system), and consists of a *benchmarking procedure*, a *workload*, a *vulnerability injector* and a *vulnerability library*, and an *attackload* (a set of attacks execute against the system under test). The authors in (Neto and Vieira 2011) proposed a different approach to security benchmarking, by assessing the trustworthiness (i.e., the accumulation of evidence that something can be trusted) of web applications and systems. Differently from security benchmarks, the goal of a trustworthiness benchmark is to increase the thrust in security attributes of a system or parts of it. The benchmarking procedure involves the analysis of the code of a specific system or component by using static code analyzers (SCA), which results in a number of vulnerabilities reported (NVR) that is used to estimate trustworthiness.

Benchmarking frameworks are lacking in the failure prediction scenario. In this direction, benchmarks for machine learning models can be adapted to the failure prediction problem, even if only some models can take advantage of the existing approaches. **Benchmarking machine learning models** is a well-known problem in the machine learning community, typically addressed by using well established *datasets* (see e.g., (Zheng 1993; Maxion and Tan 2000)), which correspond to the *workload* mentioned above. The datasets include data generally accepted by a community (e.g., IRIS dataset and others (Bache and Lichman 2013)) that the tool or algorithm must process to assess its performance (prediction accuracy, recognition error rate, etc.). These datasets can be used independently of any system configuration. However, as mentioned before, such repositories are not enough for assessing and comparing failure prediction algorithms on a particular system, as the data may reflect the behavior of the several different systems.

2.7 Final remarks

This chapter presented background on dependability concepts for computer systems, online failure prediction and fault injection techniques, including the state-of-the-art in such areas.

Online failure prediction is a novel technique that allows detecting in advance the occurrence of failures and to mitigate their effects. The review of the literature highlighted the fact that the approaches proposed in the past are seldom used in commercial system. This is mainly due to the fact that failure models are complex to train and optimize, and failure data are usually not easy to collect. The solutions proposed in Chapter 3 and Chapter 4 are in line with this need, where fault injection is used to generate failure data to train and optimize failure prediction models on particular system installations.

Another key aspect regarding the use of failure prediction models is the evaluation of the prediction performance, which requires rigorous procedures and metrics. We address such need with the proposal of a benchmark for failure prediction models in Chapter 5. The benchmark is based on the failure-data generation approach proposed in Chapter 3 and defines the components and procedures needed to assess and compare different models.

The chapter also introduced several failure prediction models (whose management is mostly manual) applied to complex computer-based systems. However, although complex software systems tend to evolve and change, very few works focusing on the adaptation of failure prediction systems can be found in the literature. This highlights the need for a framework that allows the continuous adaptation of online failure prediction systems, as the one presented in Chapter 6.

Finally, the key problem of optimizing failure prediction models has been addressed in few works, especially regarding what concerns the selection of the most adequate variables to predict failures. In fact, works in the literature are limited to *a priori* analysis of system characteristics that can help in predicting failures, and experimental evaluations based on few failure data. Chapter 7 presents the study of the application of a symptoms identification approach for facilitating the selection of a set of variables for an optimal failure prediction.

Chapter 3
Generating failure data by Software Fault Injection

Predicting failures in computer systems is possible by modeling the behavior of the system in the time instants preceding the occurrence of failures. Failure prediction models may be built manually (which is rare, due to the complexity of such task) or by using *training* (or *learning*) *algorithms*. The aim of training algorithms in the failure prediction context is to take data monitored from a *target system* (e.g., page faults per second, I/O request queue size, etc.) and relate them to observed failure events in the form of a model. In this scenario, **failure-related data** are needed to train the prediction model and optimize its performance, as well as to validate the accuracy of predictions.

As failures are rare events, data collection usually takes a long time (e.g., (G.A. Hoffmann 2006; M. Vieira et al. 2009; Bao, Sun, and Trivedi 2005), which limits the applicability of failure prediction. Different solutions addressing such limitation were proposed, including the use of existing failure data, often collected and stored in collaborative repositories (hosting failure data from several systems), as for instance the repository at (Usenix and Carnegie Mellon University (CMU) 2006). This solution has a clear limitation, as a system may evolve over time (leading to the need for new failure data to be collected) or the collected failure-related data may come from systems with different characteristics. In fact, it is fundamental that the failure data represents the relation between failure events and the dynamics of the concrete target system (i.e., the system where failure prediction is being implemented), and that depends on specific properties of the individual components of the system and/or of the system as a whole, which may vary from one installation to another (e.g., a software version upgrade may impact the system behavior).

In this chapter we propose a practical approach for **generating failure data based on the injection of realistic software faults in a specific target system**. The reasoning is that injecting faults increases the probability of a system to fail, hence enabling a fast generation and collection of failure-related data. The ultimate goal is to facilitate the use of failure prediction models on specific systems by addressing the problem of failure data scarcity. We believe that the use of software fault injection for generating failure data in short time facilitates and speeds up the training of failure prediction mechanisms and their optimization, among other aspects.

The proposed approach makes use of the **Generic-Software Fault Injection Technique** (*G-SWFIT*, (J. A. Duraes and Madeira 2006)), the *de facto* standard for emulating software faults in a representative way. In practice, G-SWFIT defines a model of the faults to be injected to emulate residual (realistic) software faults, based on an extensive field study (J. A. Duraes and Madeira 2006). The technique supports the injection of software faults by modifying the target system's code at machine-code level, instead of modifying the system's source code, a key feature when source code is not available, and especially for nowadays' widely used OTS-based systems. The reason that stands behind the choice of injecting realistic (or representative) software faults stands in the fact that software faults are nowadays the largest cause of computer system failures (Lee and Iyer 1995; Kalyanakrishnam, Kalbarczyk, and Iyer 1999) and that representativeness is a fundamental property for assuring that the generated failure-data can be used in practice (J. A. Duraes and Madeira 2006). As mentioned before, the injection of software faults was first addressed by Christmansson in (Christmansson and Chillarege 1996), followed by many other works (e.g., (Hsueh, Tsai, and Iyer 1997; Aidemark et al. 2001; J. A. Duraes and Madeira 2006; Carreira et al. 1998), among others), resulting on several fault injection techniques with different purposes and targets.

In the **failure prediction context**, the injection of faults has been considered for assessing failure prediction and/or detection mechanisms. For instance, Gross et al. (Gross, Bhardwaj, and Bickford 2002) emulate memory leaks in an Apache web server by modifying the source code in a way that prevents objects from releasing memory space at the end of their lifecycle. The same solution was used by (Alonso, Torres, and Gavaldà 2009) to validate mechanisms for the detection of resource exhaustion in a Tomcat web server. However, the aim of fault injection in those works was to assess systems against well-known fault types (e.g., memory leaks), while there is no study on the use of failure data generated by fault injection for training and assessing failure prediction mechanisms.

The proposed approach includes a method for assessing the **accuracy of the (synthetic) failure data generated** with respect to failure data that would be collected in a real scenario. In this perspective, we consider that, although realistic fault injection is a necessary condition to generate realistic failure data, it may not be a sufficient condition and thus the quality of the generated data must be assessed. In particular, we study the conditions under which failure data generated with our approach can be accurate, and propose a set of metrics for estimating such property. The analysis of this property is a mandatory aspect to enable the use of the failure prediction models in a real scenario with a known level of confidence.

The chapter is organized as follows. Section 3.1 introduces the approach for generating failure-related data using fault injection. Section 3.2 describes the first phase of the approach, namely the definition of the failure data generation environment, which includes the specification of the failure(s) to predict and the faults to inject, among others. Sections 3.3 and 3.4 describe the core of the approach, presenting how to generate, collect and organize the failure data. Section 3.5 introduces the solution for assessing the accuracy of the failure data generated. A case study is presented in Section 3.6, where we discuss the training, testing and assessment of a novel failure prediction model running on a Windows XP OS environment. Finally, Section 3.7 concludes the chapter.

3.1 Overview of the approach

The approach for generating failure data includes an **experimental procedure**, a set of **components** for controlling the fault injection process and the dataset building, and a **method for estimating the accuracy** of the generated data. Software faults are injected while the target system executes one or more operations (a group of these is called a *workload*), in a way that allows capturing the dynamics that lead to failures by monitoring several variables (numerical data, events, etc.). In practice, the approach includes the following components:

- **Fault injector and faultload**: faults are defined and organized in a faultload. A fault injector emulates specific faults by modifying one or more components of the target system. The choice of the faultload is of utmost importance as it influences the data generated, ultimately impacting on the overall results (different faults may lead to different types of failures).
- **Workload**: for collecting information about the system behavior, faults must be injected while the target system runs a workload, and this procedure should be repeated several times. The workload is the set of operations that the target system performs in the field (realistic workload) or, alternatively, it may be a set of synthetic operations (a synthetic workload) that represents the usual tasks of the system, built specifically for failure data generation and collection. A synthetic workload is useful when the system has not been deployed yet, or when it is not possible to inject faults in the target system and/or the workload cannot be replicated.
- **Monitoring and data collection infrastructure**: an infrastructure is used to gather the data that characterizes the behavior of the target system in the context of the observed failure events, while running a workload and

injecting faults. Depending on the failure prediction mechanisms under study, besides failure-related data, one may need to collect also failure-free data. What is important is the data to include only the most relevant information for predicting failures.

The components above are the fundamental parts of the **experimental procedure**, which is divided in *four phases* (see also Figure 3.1):

1) **Definitions and set-up**: in this phase one must define the failures to predict, the system information to be monitored (e.g., a set of numerical variables or a set of events in the logs, including failure events), the workload and the faultload, and a set of parameters characterizing the scope of the failure prediction. This comprises building the concrete faultload to inject, installing and configuring the workload emulation tool, and installing and configuring the data monitoring and collection infrastructure and the fault injection tool. Other tasks include defining and setting up the *target system*, i.e., the system where failure prediction will be implemented), and a *controller* system, independent from the target system, for controlling the experiments and collecting the failure data.
2) **Data generation and collection**: this is the core phase of the approach, where the data are collected while the target system executes the workload and faults are injected by a tool implementing the G-SWFIT recommendations (J. A. Duraes and Madeira 2006). This data may correspond to fault-free situations (Golden Data) and/or situations in which a failure is observed (Failure Data). Data collection is done during several time intervals and in each interval the monitoring infrastructure collects the values of the variables portraying the state of the target system.

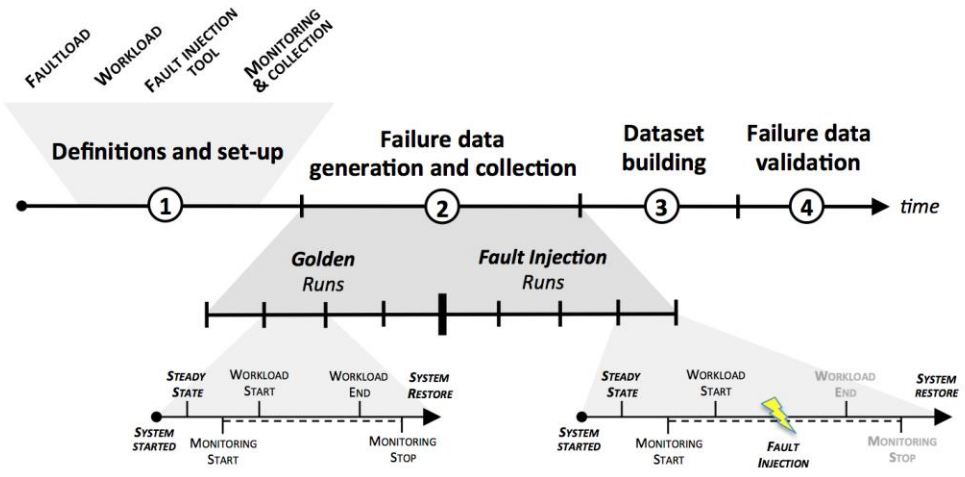


Figure 3.1 – The four phases of the failure data generation

3) **Dataset building**: the data collected are organized in datasets for being consumed later by the failure prediction models. This process depends on the failure prediction system to be trained (e.g., training anomaly detection systems only requires Golden Data), as well as on the types of failures being predicted. In particular, the monitored data are associated with the failures observed in Phase 2 considering the failure prediction parameters specified in Phase 1.

4) **Failure data accuracy estimation analysis**: accuracy is the property of the generated failure data to be similar to data that would be obtained in a real scenario. Due to the scarcity of real data, we estimate the correlation between synthetic and real failure data by applying metrics (specific of each condition) to two or more, independently generated, synthetic failure datasets. We use the concepts of *weak accuracy* and/or *strong accuracy*, as sufficient conditions for the generated failure data to be considered accurate. Strong accuracy metrics are applied directly on the datasets, while the weak accuracy metrics are applied to the prediction performance of the models trained with independent synthetic datasets.

3.2 Phase 1: Definitions and set-up

The goal of the first phase is to specify the data generation environment and the scope of the failure prediction task (e.g., the type of failures to predict, the prediction advance, etc.), and to set-up the fundamental components (faultload, workload, fault injection tool, etc.). In practice, this includes:

1) Defining the **types of failures to predict**, considering the different failures that may affect the target system. This can be based on historical information, on taxonomies of common failure modes (e.g., the C.R.A.S.H. scale (Koopman et al. 1997)), or on the identification of system-specific failures (e.g., service degradation);

2) Defining **failure detectors** (models) able to detect the failure events that should be associate with the monitored data, which will then be identified as failure-related data;

3) Selecting the **software faults to inject**, which represent the root cause of the failures observed on the target system. A generic faultload is defined in (J. A. Duraes and Madeira 2006), but specific faultloads may also be considered;

4) Installing a **software fault injection tool** that implements the G-SWFIT recommendations for injecting the software faults at the machine-code level;

5) Defining the **workload to use**, which must emulate the target system's operations during the fault injection process;

6) Selecting the **system variables to monitor** that will be correlated with the observed failures. These variables may be selected using specific techniques, methods or algorithms, such as feature selection;

7) Scoping the **failure prediction problem**, which includes defining the problem of predicting failures according to given models or frameworks, as for instance the time to failure, the prediction window, the probability of a failure to occur in a given time interval, and so on.

The **environment for generating failure data** includes a target system (the system on which the failure prediction will be performed) and a controller (a machine that manages the fault injection, the failure data collection, etc.). This separation is not strictly needed, but it reduces intrusion in the target system and does not influence the failure data generation process. In practice, the controller is in charge of controlling the target system (e.g., boot, reboot, restore a fault-free state[5]), the injection of the software faults, the workload (e.g., execute, stop) and the collection of the failure-related data (which are stored in a local database). The parameters that define the data generation (e.g., the number of fault injection runs to perform, the time horizons to consider, etc.) are also managed by the controller machine, but are defined in Phase 2 of the approach (see Section 3.3). On the other hand, the target system executes the workload, and hosts the fault injection tool and the monitoring tool. Figure 3.2 presents the distribution of the fundamental components on the target and controller systems.

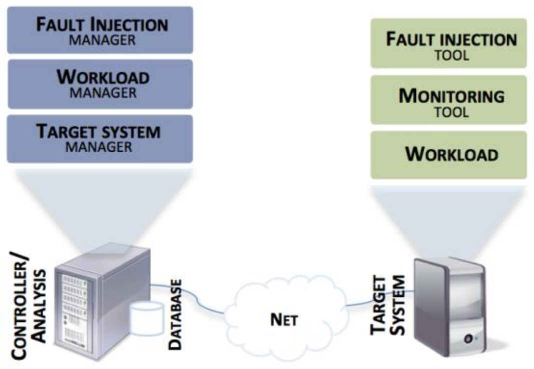


Figure 3.2 – The failure data generation environment

[5] We here use of the term "fault-free" for indicating a status of the target system in which *no fault was injected*, and not for referring to the ideal status in which the target system is free from any software fault.

3.2.1 Characterizing the failures

The first step towards failure prediction is to be able to correlate the monitored data with the failures observed during fault injection, which requires a precise definition of what a failure is. Different types of failures may occur during the data generation and collection phase, thus recognizing failure events and their occurrence time is required, which can be performed by means of tailored failure detectors.

In general, a **failure** is an event that corresponds to the interruption of the correct functioning of the system or one of its components. In the direction of categorizing failures, we can find several works such as the classification from (Avizienis et al. 2004), the C.R.A.S.H. scale (Koopman et al. 1997), the distributed fault model (here fault is intended as failure) (Tanenbaum and Van Steen 2007), and the generic failure model by (Bondavalli and Simoncini 1990). The failure model proposed in (Avizienis et al. 2004) can be considered the most general one as it is based on the other works listed. In this work we do not propose any particular failure mode classification, although we do recommend the definition of the failures to predict according to the classification given in (Avizienis et al. 2004).

Failure detectors are tools based on models that recognize a failure occurrence, by identifying failure patterns in the target system. As the accuracy of such models may impact the quality of the generated data, thus affecting the study of failure predictors, the quality of failure detectors should be assessed and their performance should be optimized. Although the definition of failure detectors and their optimization are out of the scope of this work, we here give an insight on how they can be implemented. In practice, failure detectors can be built in two ways:

1) *Manually* by defining a set of conditions or rules for recognizing a failure in the target system (e.g., if the system does not respond to a ping for more than a minute, this means that there is a crash or a hang). This type of modeling is possible when failures can be easily described by using simple rules, either considering common failure modes (e.g., crash, hang) or service specific failures (e.g., performance failure);

2) *Automatically*, in the case of failures events not easy to describe, by using machine-learning algorithms that are applied to a set of collected data. This type of modeling is necessary when, for instance, the detection of a failure is influenced by dependencies between several variables (e.g., OS-level hangs, involving complex interactions between OS-level and user-level components (Antonio Bovenzi et al. 2011)).

A failure detector must also identify the occurrence time of a detected failure (referred to as **failure time** T_F), which may be a non-trivial task, as it depends on the type of failures being addressed, among many other aspects (e.g., the time when an *Hang* is detected often does not correspond to the instant in which it actually occurred). However precise solutions to this problem are out of the scope of the present work, where we just recommend adopting a best-effort approach, considering the failure time as the detection time. Obviously, the better the detection system is, the more precise is the failure time estimation.

3.2.2 Defining the faultload and the fault injection procedure

The faults to be injected (***what***) must be carefully chosen and correctly emulated in order to be *representative* of *residual software faults*, i.e., software faults that are likely to escape the testing phase and be present in the target system (J. A. Duraes and Madeira 2006). The need for the software faults to be realistic stands in the fact that such faults are more likely to lead to failures similar to the ones that would occur if real faults affect the system, thus allowing generating realistic failure data.

Choosing the types of faults to inject is a non-trivial problem. The authors of the G-SWFIT proposal (J. A. Duraes and Madeira 2006) identified three conditions for emulating realistic software faults, namely: i) choose the **types of faults** likely to exist in the target system; ii) reproduce **patterns** that represent software faults present in software systems; and iii) inject faults according to a given **distribution**, in order to mimic residual faults. (J. A. Duraes and Madeira 2006) analyzed software defects in several open-source software products that were corrected from a version to another (a procedure similarly to the one conducted for building ODC: see Section 2.4.3), but focusing on the fact that faults can be due to missing, wrong or extraneous (in excess) programming language constructs (e.g., statements, expressions, function calls, etc.), which supports the definition of *what* to inject. In practice, the resulting dimensions (or classes) of such model also define **how to emulate a given type of fault**, classifying the faults in terms of programming language constructs (e.g., statements, expressions, function calls, etc.) that can be *missing*, *wrong* or *extraneous*. In this work, we adopt the fault model proposed in the context of the G-SWFIT technique (J. A. Duraes and Madeira 2006), already presented in Table 2.3 (Section 2.4.4), as the fault types defined provide a fine-grain classification that can be easily translated into a code mutation (see Table 3.1 for some examples). Furthermore, such fault model includes the distribution of the occurrence (or *presence* rate) of the faults, which is of utmost importance as it allows defining a more representative distribution of the types of faults to be injected. For instance, MIFS faults can be considered representative of real scenarios if their weight in the total number of faults injected is about 10%.

Regarding the injection location (***where***), the rules for identifying where a specific type of fault can be emulated are also defined in (J. A. Duraes and Madeira 2006), and are referred to as *"patterns identifying an injection location"*. In practice, an injected software fault is a mutation of the code in the specified location(s), achieved by applying a given *fault operator*. Table 3.1 presents three examples of fault operators for (a) a *"missing local variable initialization (MVI)"*, (b) a *"missing function call (MFC)"*,

Table 3.1 - G-SWFIT: examples of mutation and search patterns

Fault type	Patterns	Code mutation/ Fault operator
(a) Missing local variable initialization (MVI) pattern and mutation	*local_var = value*	*remove local_var initialization*
	local_var = some_other_variable	
	local_var = expression	
(b) Missing function call (MFC) pattern and mutation	*function_name(…);*	*remove function call*
(c) Wrong value assigned to a variable (WVAV) pattern and mutation	*local_var = value*	*local_var = other_value*

and (c) a "*Wrong value assigned to a variable (MVAV)*". The column "*Patterns*" presents the situations in which the fault can be injected and the column "*Code mutation/Fault operator*" identifies the operation needed to emulate the fault type. For example, the locations in which a MVI can be injected are identified by the presence of a variable initialization instruction (e.g., *local_var = value*) and the fault is emulated by removing the initialization instruction.

It is worth noting that, although the fault patterns defined characterize the potential injection locations, there may be many eligible locations and injecting faults in each and every one may not be feasible and/or representative. Although the problem of selecting code locations is out of the scope of our thesis, we follow a simple rule of injecting faults in highly executed software components, which can be easily identified by using profiling tools (i.e., running the workload on the system and identifying the components that are most executed (Ball and Larus 1994)). This increases the fault activation probability, which potentially leads to the observation of a reasonable number of failures. However, the representativeness of the faults injected in such widely executed locations must be taken into account. In fact, as defined in (R. Natella et al. 2010), injected software faults are representative if their location is rarely executed or they are rarely activated when their code location is executed (otherwise, they would be easily caught during tests). Hence, low activation rates are a required condition for fault representatives and should be experimentally verified. (R. Natella et al. 2010) defines 5% (i.e., 5% of the faults injected lead to failures) as an acceptable activation rate for representative software faults.

About the time (***when***), the G-SWFIT proposes the injection of software faults in the target system's compiled code, both for representativeness and generality reasons. In fact, injecting in the compiled code means that the *code mutation* tries to represent a *residual fault* caused by a programmer error, which is translated by the compiler (as in real scenarios). The G-SWFIT technique is *generic* in the sense that faults can be injected even when the system's source code is not available, which is a usual requirement when dealing with third party or OTS-based systems.

In summary, from the perspective of generating failure data, the faultload must consider the scope of the target types of failures, as usually the potential target code is too big for faults to be injected in all possible locations. For example, crash failures can be caused by emulating a segmentation violation by injecting a WVAV (*wrong value assigned to a variable*) in core functions of the operating system. In practice, one must define a proper policy for choosing the most representative faults to inject, with the aim of increasing the probability of failures, a potential policy is "*choosing locations in the most executed parts of the software*".

3.2.3 Defining the workload

A workload can be seen as a collection of programs, data and commands used to exercise a system (D. Ferrari 1972; Domenico Ferrari 1984). In practice, failure-related data represent the failing behavior of a target system when executing a concrete workload that leads to the activation of injected faults by exercising the functionalities of the system.

The generation of failure data during the operational phase of the target system can make use of the real workload that the system has to execute. However, the most general case is to perform the data generation before deploying the system or using a copy of the target system (as proposed in Chapter 4), as injecting faults during system operation may not be feasible, due to possible damages to the software, the data, the environment, etc. In such cases, it is necessary to define a specific workload to exercise the system.

The choice of the most adequate workload is very dependent on the target system. In general, a workload must be chosen or defined in a way that reproduces the typical behavior of the system and the use of a particular workload strictly depends on what is available about it. In practice, a workload can be of three types: a **real** workload, a **realistic** workload, or a **synthetic** workload. Real workloads are made of actual applications used in real environments. Results using real workloads are quite representative, but access to them is frequently not possible. On the other hand, realistic workloads are artificial workloads that are based on a subset of representative operations performed by the system. Although artificial, realistic workloads reflect the real situation, and are still quite representative and easier to implement. Finally, a synthetic workload can be a set of random operations, and is easier to define, but obviously results may have a low representativeness.

A potential way to build realistic workloads is to collect, study and classify the different types of operations executed by the target system, and then mimic them using a custom made application. Workloads from standard benchmarks (e.g., SPEC, TPC, etc.) typically offer realistic workloads, each one relative to a particular system domain. For instance, TPC-W and TPC-App offer workloads for transactional web-serving systems (“Transaction Processing Performance Council (TPC)”), while SPEC benchmarks provide workloads for assessing CPUs, Workstations, Virtualization systems, etc. (“Standard Performance Evaluation Corporation (SPEC)”).

We do not propose any particular methodology for building a proper workload, as this is not central to the present work. However, several methodologies can be found in the literature (e.g., (Agrawala, Mohr, and Bryant 1976; Calzarossa, Italiani, and Serazzi 1986; Calzarossa and Serazzi 1993; Eeckhout et al. 2005; Moro, Mumolo, and Nolich 2009)), including in the fault injection field (Cotroneo, Fucci, and Natella 2012; A. Bovenzi et al. 2011).

3.2.4 Selecting the variables and the monitoring infrastructure

Online failure prediction models are built from observations about the past behavior of a target system (or the evolution of its inner *states*), which can be described by **numerical time series data** or **categorical data** (e.g., events stored in log files). As online failure prediction models forecast failure events by comparing the observed behavior with the evolving target’s state, their efficacy and accuracy is dependent on the quality of the observations (i.e., the data) collected from the target system.

A computer-based system, as well as each of its components, may be described in several ways, which results in a large number of variables that can be monitored. This problem is known as *feature extraction*. In practice, the features or variables that better characterize the behavior of a computer system are usually defined based on the experience of users and developers and describe a specific behavior of the whole system or of a part of it. This approach is different from the ones followed in other areas, e.g., computer vision and image segmentation, in which a subject (e.g., the image of a human face) may be automatically analyzed for extracting information or variables for achieving a given goal (e.g., recognize a subject by searching a specific set of characterizing features). In this work we assume that the best features are among the ones a computer system makes available, using *de facto* the Ockham’s razor principle (Gernert 2009; Blumer et al. 1987). The existing tools for monitoring computer systems provide a finite set of variables, each one representing a characteristic of the system (e.g., the available free memory in MBs, the number of page faults/sec), and thus limit the features that can be extracted.

The set of variables that is more adequate for prediction is not known *a priori* and several steps are needed to reach an optimal set. The selection or identification of the most adequate set of variables is also known as **feature selection problem**, which must address several aspects. For instance, models based on too few or too many variables may equally lead to poor performance (Baum and Haussler 1989; Geman, Bienenstock, and Doursat 1992; Hochreiter and Obermayer 2006; G.A. Hoffmann, Trivedi, and Malek 2006). In addition, considering that the variables form an *n*-

dimensional space, the modeling of a specific reality identified by a set of variables (as the prediction of a specific event) is computationally harder as the number of dimensions *n* increases (the performance of a model degrading as the dimensionality increases is also known as the *curse of dimensionality*, or Hughes effect (G. Hughes 1968)). Selecting an unfavorable subset of variables can lead to poor modeling performance as well. This way, it is of utmost importance to select the smallest set of variables that allows achieving the best prediction performance: (G.A. Hoffmann, Trivedi, and Malek 2007) analyzed the impact of variables on different models and demonstrated that the choice of the variable set has a strong influence on the prediction performance. The same authors showed that variables chosen by experts are likely to not form optimal sets for failure prediction and resource forecasting (e.g., used in the software aging detection context).

Several techniques can be used to perform feature selection (see Chapter 2), which can be divided in two groups: the **filter** and the **wrapper** approach. In the former (*filtering*) the feature selector filters the irrelevant attributes and is *independent* from the specific model that will use the variables. In the latter (*wrapping*), the most important features are filtered in the context of the prediction model and taking into account the specificities of the underlying learning mechanism. In addition to these two, the *hybrid* approach tries to take the advantages of both *filtering* and *wrapping*.

In this work we propose the use of **feature selection techniques**, although we do not specify any particular one. In addition, we recommend the use of numerical data instead of categorical data (e.g., data stored in logs), as it has been demonstrated that categorical data may degrade the performance of a prediction model (G.A. Hoffmann, Trivedi, and Malek 2007). For example, (G.A. Hoffmann, Trivedi, and Malek 2007) compared the performance of a model for predicting failures in a complex telecommunication system, first trained using numerical variables and then trained using numerical and categorical variables (G. Hoffmann and Malek 2006). Results show that the addition of log data degraded the model's prediction performance. However, categorical data that can be transformed to discrete-time numerical variables may be of interest. We also propose to normalize the collected numerical variables, as recommended by several works in the machine learning area (G. Hoffmann and Malek 2006).

Several **monitoring tools** for collecting the values of variables over time can be found in computer systems. Well-known solutions are the Linux/Unix command *top* (showing information about system's CPU usage, Memory usage, Swap memory, Cache size, processes, etc.), the *proc/stat file* (number of processes executing in user mode or kernel mode, jobs waiting I/O to complete, etc.), and open-source tools such as NMon (generic system information), Nagios (network and server-related information), both for Linux and Microsoft Windows computer systems, and Logman included in the Microsoft Windows OS.

Data must be collected according to a specific **sampling rate** that depends on the monitoring system used and the information being collected. Typically, operating systems provide standard sets of variables to monitor, representing a value in a given time instant (e.g., amount of free memory) or a rate (e.g., number of CPU Level 1 cache-misses per second). The monitoring system should be able to manage such information, i.e., the increment of data, its storage, etc. The analysis of the optimal data sampling rate is out of the scope of this work, although few works in failure prediction field (e.g., (G.A. Hoffmann 2006)) demonstrate the influence that such parameter can have on the failure prediction accuracy.

3.2.5 Modeling the failure prediction problem

After defining the failures to predict, one must characterize the failure prediction approach, which will serve for the definition of the datasets to be consumed by the failure predictors, including the association of the failure instant to the data collected and their labeling. In practice, such characterization includes identifying and defining the key characteristics that should be taken into account when predicting a failure (e.g., the expected failure time, the distribution of failure probability over a given time interval, etc.) and allows associating a failure event to the data by labeling each data sample (see Section 3.4). This task is named **failure prediction problem modeling**.

The literature on failure prediction is abundant in what concerns algorithms and prediction systems, although such works do not share a model for predicting failures, thus resulting in no common definitions. However, besides providing a survey on the existing online failure prediction systems, (F. Salfner, Lenk, and Malek 2010) propose a generic model for addressing the online failure prediction problem. As shown in Figure 3.3 the failure prediction task consists of assessing if, at a time *t*, a failure is going to occur within a precise time, called *lead-time* Δt_l. The prediction can be valid in a time window, named *prediction window* Δt_p. The variation of the parameters Δt_l and Δt_p influences the performance of the prediction. In practice, at time *t*, a model (or predictor) should predict if a failure is going to occur in the interval $[t+\Delta t_l, t+\Delta t_l+\Delta t_p]$.

Although other models can be used, **in this work we adopt and include Salfner's model in the experimental process**, as it allows a complete representation of the failure prediction event, its expected arrival time, as well as the modeling of the

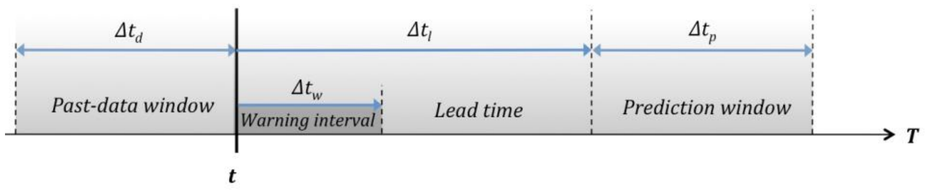


Figure 3.3 – Time relations in Online Failure Prediction

minimal prediction time (or prediction convenience time) and the amount of data needed for performing the prediction (e.g., training failure prediction model). More details about this framework can be found in Section 2.2.3.

3.3 Phase 2: Data generation and collection

This phase consists of combining the components of the approach (faultload, workload, monitoring tool, etc.) to generate failure-related data for training, assessing and improving failure prediction models on a particular target system. In practice, after the definitions phase (Phase 1) in which the target system is installed and the environment is set up, the data generation and collection takes place by implementing a procedure that includes several time intervals (as shown in Figure 3.4), referred to as runs, during which the monitoring infrastructure collects the set of variables selected. The **number of runs**, as well as their **duration**, depends on several parameters, such as the time needed to execute the workload, the specific set-up environment and the prediction parameters (e.g., for predicting a failure one hour in advance, each run must last for at least one hour). Depending on such needs, the user must define a *maximum* run execution time T_{MAX}, which obviously has to be greater than the workload execution time T_W.

Failure data are data obtained by injecting faults during several runs (eventually evolving into failures), while golden data are gathered when no faults are injected and no failures are observed[6]. The use of one or both kinds of data depends on the prediction model (or models) that will consume the generated data (e.g., anomaly detection based models just need golden data, while classifiers need both types of data). A run with no faults injected and no failures observed is called *golden run*, and the corresponding data are *Golden Data* (GD). An execution in which faults are injected is called **Fault Injection Run**. If a failure is observed during a fault injection run then it is a **failing run**, and the data monitored are *Failure Data* (FD). Also **non-failing runs** can exist, with associated *Non-Failure Data* (NFD). Although this kind of data may also provide information about the system failing behavior, their use is out of the scope of this work. In each failing run, the failure event must be detected and latter (in Phase 3) associated to the collected Failure Data. For this, different **failure detectors** (models that recognize failure patterns when they occur, as defined in Section 3.2.1) may be needed.

When more than one failure mode or more than one workload is considered, the runs (and thus the failure data) can be grouped into *Scenarios*. In this work, the scenarios are identified by a failure mode $\mathcal{F}$ and a workload W, or alternatively by the tuple <*Workload, Failure mode*>.

[6] In fact, no fault is injected and no failure is observed does not mean that no fault was activated, as there is not guarantee that no residual faults are present in the system.

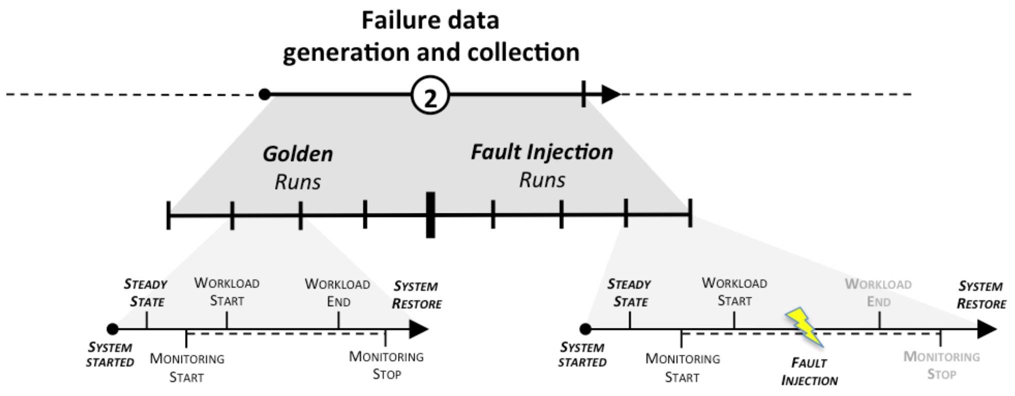


Figure 3.4 – Failure data generation, collection and data organization phases

As detailed in Figure 3.4 the data are generated as follows:

1) Each run starts by booting the target system and waiting for it to **reach a steady state**, before the workload is executed. Having the system in a steady state means that it is ready for executing the workload in the best way possible, which is recommended, albeit not mandatory. The instant in which the system achieves its steady state is referred to as T_0.

2) **The workload and the monitoring tools are then started**. The instant in which the workload execution starts is referred to as T_W, while T_M identifies the time at which the monitoring system is executed. The data collection may start at time T_M or T_W, depending on the specific needs (e.g., if data from the beginning of the workload execution are needed, the monitoring must be started before the workload). In practice, data is composed of data samples collected from the different variables at a given instant of time, according to a specific sampling rate *s*.

3) In a **Fault Injection Run (FIR)**, a fault is injected at time T_{FI} while the target system is executing the workload and the monitoring tool is collecting data. The tool implementing the G-SWFIT recommendations (J. A. Duraes and Madeira 2006) injects a fault that modifies a part of the target system at machine-level code (by modifying a file or a running process) according to the guidelines introduced in Section 3.2.2. In a **Golden Run (GR)** the system executes the workload, but no fault is injected.

4) The **run finishes** when a *failure* (T_F, FIR only) is detected (the failure detector associates the failure to the time T_F), or after the *workload* has completed its execution (T_{W_END}) or a *maximum* run execution time T_{MAX} is achieved. In such cases, two situations are possible:

a) In the case of **Golden Runs (GRs)**, if no failure is detected in the interval [T_0; T_0+T_{MAX}], the data relative to the run are considered *Golden Data* (GD_{Ri}, Golden Data relative to the *i-th* run). It is worth noting that a failure occurring in a Golden run is caused by an actual residual fault of the target system (i.e., not an injected one) and the data should also be considered as Failure Data.

b) For **Fault Injection Runs (FIRs)**, if no failure is detected in the interval [T_0+T_{FI}; T_0+T_{MAX}], the run is considered to be failure-free, and the relative data to be Non-Failure Data (NFD_{Ri}, relative to the *i-th* run). On the other hand, if a failure is detected in such interval, the collected data are considered Failure Data (FD_{Ri}, relative to the *i-th* run).

5) After completing a run (and collecting the corresponding data), the target **system must be restored** to a state in which no faults injected are present. This ranges from *rebooting*, in the cases where the fault does not permanently affected parts of the system (e.g., data or files), to the correction of fault effects (e.g., substituting files previously backed-up) or the re-installation of the entire target system[7].

It is worth noticing that the T_W (workload execution time) parameter corresponds to the embedded dimension Δt_d in the model for scoping the failure prediction task that we adopted (see Section 3.2.5), and it may vary with the prediction time horizon Δt_l, depending on the type of failure prediction model that will consume the data (e.g., reliability-growth prediction models can predict far in the future needing few data, although with a low prediction accuracy).

3.4 Phase 3: Dataset building

In this phase, the collected Golden Data and Failure Data are associated to information about the failures observed during Phase 2 and organized into **datasets**, for being later used for training and validating failure prediction models. Such association is implemented by labeling each data sample composing the collected data.

Labeling data is a technique that associates a numerical label (e.g., 0, 1, etc.) to each data sample (i.e., a set of values of each monitored variable), depending on the meaning that each label has in the particular modeling or prediction scenario (e.g., a

[7] Virtualization is a solution that allows restoring the target system (both software and – emulated – hardware), by using check-pointing and restoring operations. The potential use of virtualization for supporting the generation of failure data is addressed in Chapter 4.

sample is labeled 0 if the target system was working correctly at the moment of the sample's collection, or conversely is labeled 1 if the system was presenting an erratic behavior). In our scenario, data is labeled according to the **failure time T_F** and the **failure prediction lead-time and prediction window** ***(Δt_l, Δt_p)***, defined in the failure prediction framework adopted (described in Section 3.2.5). The idea is to consider a model that predicts a failure Δt_l time in advance, with a variation of Δt_p with respect to the failure time T_F. We must note that a successful prediction depends on the patterns that the data may show Δt_l time before a failure occurred, which can be present or not: in the worst case, the predictor will have a poor prediction performance, being not able to distinguish between failure-prone and non-failing situations. It is also worth noticing that such method for associating the information about the prediction of a failure to the data is not unique (e.g., regression models trained on specific lead and prediction times can also be used). Nevertheless, we adopted labeling for its generality and because it facilitates the study of the relationship of each data sample with the failure observed.

The labeling of the collected data is performed as follows. Data from a given run r is composed of n different variables $\underline{v}^r = <v^r_1, v^r_2, \dots, v^r_n>$, where v^r_i is the *i-th* variable collected from the target system (Figure 3.6 (a)). For each time instant k, each variable v^r_i has a given value $v^r_i(k)$, representing a variable value collected at the time instant k. Hence, a *data sample* at time k is defined as:

$$\underline{v}^r(k) = <v^r_1(k), v^r_2(k), \dots, v^r_n(k)> \quad (3.1)$$

A *data sample* $\underline{v}^r(k)$ collected during a *Golden Run* (when no failure occurred), is associated a **label** $l^r(k)=0$, for each time k. On the other hand, given T^r_F the time at which a failure was detected during the Failure Run r, and the prediction indexes $(\Delta t_l, \Delta t_p)$ (valid for all the runs), a label $l^r(k)=1$ is associated to a data sample $\underline{v}^r(k)$ if a failure occurred in the interval $[T^r_F-(\Delta t_l+\Delta t_p), T^r_F-\Delta t_l]$, otherwise it is 0[8]. Hence, for each time instant k and each run r, a labeled sample is:

$$\underline{v}^{r*}(k) = <v^r_1(k), v^r_2(k), \dots, v^r_n(k), \mathbf{l^r(k)}> \quad (3.2)$$

The collected data labeled according to the failure prediction indexes $(\Delta t_l, \Delta t_p)$ and the failure time T^r_F can be considered a **dataset**. More generally, several couples $(\Delta t_l, \Delta t_p)$ can be specified, and varying the values of Δt_l and Δt_p let the labels associated to each data sample to change accordingly. In this case, being $\underline{\Delta t_l}=<\Delta t_{l1}, \Delta t_{l2}, \dots, \Delta t_{lL}>$ and $\underline{\Delta t_p}=<\Delta t_{p1}, \Delta t_{p2}, \dots, \Delta t_{pP}>$, one can define a dataset with which ***N* sets of labels** are associated, where $N = |\Delta t_l| \mathrm{x} |\Delta t_p|$. Of course, the Golden Data will present 0s for all the values of the couple $(\Delta t_l, \Delta t_p)$. Such dataset can be built once and used for training and testing a failure prediction model using a couple $(\Delta t_l, \Delta t_p)$ at a time.

[8] It must be noted that the label values chosen can be any two different numerical values (other widely used values for labelling data are (-1, +1) – especially when using Support Vector Machine classifiers – (5, 10), and so on).

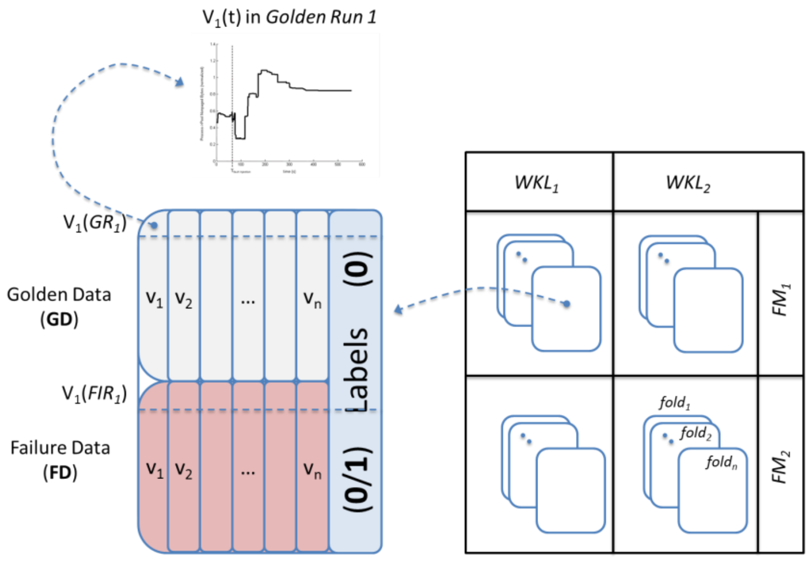


Figure 3.5 – Datasets and scenarios (two workloads and two failure modes)

The collected and labeled Golden and Failure Data from each run are then organized in a **global dataset**. Each scenario <*Workload, Failure mode*> is associated to a given global dataset, as data reflect different failure modes and workloads (see Figure 3.5).

An example of a dataset is presented in Figure 3.6 (a) and (b): Figure 3.6 (a) represents data collected from the *i-th* Fault Injection Run and labeled with *N* different couples of *(Δt_l, Δt_p)* values, while Figure 3.6 (b) presents a global dataset, highlighting the difference between labeling Golden and Failure Data, being the first labeled with only 0s and the latter with 0s and 1s.

For training and validating failure prediction models, each global dataset should be divided in **training datasets** (**TDSs**) and **testing datasets** (**TTDSs**), whose goal is to support the assessment of prediction performance. Such division is usually based on grouping single *data samples*. However, in our work we group Golden and Failure data in training and testing datasets by considering the *runs* to which they belong to, thus implementing a **run-by-run data partition**. The reason that stays behind this decision is that the collected data represents time series and the division in *samples* may alter the *continuity* and *ordering* among samples, which may finally impact the prediction performance (Dietterich 2002) (e.g., when training regression models).

The division in training and testing sets should be done according to a certain policy (see e.g., (Vapnik 2000)): for instance, half of the data samples are used for training and the other half for testing. In practice, one must consider the fact that training a failure prediction model with a small percentage of data may result in a high variance of the prediction performance.

On the other hand, using a high percentage of data for training may also result in poor prediction performance. In fact, in such case the predictor is trained for predicting an event according to a very specific pattern without considering the possibility of the pattern to suffer small variations in a different scenario. Such problem is called *overfitting*, and calls for a predictor to *generalize* its predictions.

A solution to the generalization problem is to conduct a validation of the model, which consists of computing how much the predictions of a model generalize to an independent dataset. Validation is widely used for different prediction models (Dietterich 2002) by analyzing the variation of the predictor's performance (in practice, by estimating of how much the performance may vary when using a dataset different from the testing dataset). Several validation techniques can be used, including for instance a simple division of the global dataset in three parts, a training

	v_1	**v_2**	**v_3**		**v_n**	**Labels$_1$**	**Labels$_2$**	**...**	**Labels$_N$**
	$v_1(1)$	$v_2(1)$	$v_3(1)$	…	$v_n(1)$	0	0	...	0
	$v_1(2)$	$v_2(2)$	$v_3(2)$	…	$v_n(2)$	0	0	...	0
	$v_1(3)$	$v_2(3)$	$v_3(3)$	…	$v_n(3)$	0	0	...	0
	$v_1(4)$	$v_2(4)$	$v_3(4)$	…	$v_n(4)$	0	0	...	**1**
	…	…	…	…	…	…	...	...	...
FIR$_i$	$v_1(k)$	$v_2(k)$	$v_3(k)$	…	$v_n(k)$	0	0	...	**1**
	$v_1(k+1)$	$v_2(k+1)$	$v_3(k+1)$	…	$v_n(k+1)$	**1**	0	...	**1**
	$v_1(k+2)$	$v_2(k+2)$	$v_3(k+2)$	…	$v_n(k+2)$	**1**	0	...	**1**
	$v_1(k+3)$	$v_2(k+3)$	$v_3(k+3)$	…	$v_n(k+3)$	**1**	**1**	...	**1**
	…	…	…	…	…	…	...	...	...
	$v_1(T_F)$	$v_2(T_F)$	$v_3(T_F)$	…	$v_n(T_F)$	**1**	**1**	...	**1**

(a)

	v_1	**v_2**	**v_3**		**v_n**	**Labels$_0$**
	$v_1(GR_1)$	$v_2(GR_1)$	$v_3(GR_1)$	…	$v_n(GR_1)$	**0**
	$v_1(GR_2)$	$v_2(GR_2)$	$v_3(GR_2)$	…	$v_n(GR_2)$	**0**
Golden Data	$v_1(GR_3)$	$v_2(GR_3)$	$v_3(GR_3)$	…	$v_n(GR_3)$	**0**
	…	…	…	…	…	…
	$v_1(GR_G)$	$v_2(GR_G)$	$v_3(GR_G)$	…	$v_n(GR_G)$	**0**

						Labels$_1$	**Labels$_2$**	**...**	**Labels$_N$**
	$v_1(FIR_{f1})$	$v_2(FIR_{f1})$	$v_3(FIR_{f1})$	…	$v_n(FIR_{f1})$	**0/1**	**0/1**	…	**0/1**
Failure Data	$v_1(FIR_{f2})$	$v_2(FIR_{f2})$	$v_3(FIR_{f2})$	…	$v_n(FIR_{f2})$	**0/1**	**0/1**	…	**0/1**
	$v_1(FIR_{f3})$	$v_2(FIR_{f3})$	$v_3(FIR_{f3})$	…	$v_n(FIR_{f3})$	**0/1**	**0/1**	…	**0/1**
	…	…	…	…	…	…	…	…	…
	$v_1(FIR_{fF})$	$v_2(FIR_{fF})$	$v_3(FIR_{fF})$	…	$v_n(FIR_{fF})$	**0/1**	**0/1**	…	**0/1**

(b)

Figure 3.6 - Data from a single Failure Run i (a) and a complete (global) dataset (b)

dataset, a testing dataset, and a validation dataset (an example of its use in failure prediction is presented in (G.A. Hoffmann, Trivedi, and Malek 2007)). The leading idea is that if the variation in the prediction performance between the testing and the validation is small, the training set is optimal for the predictor. A more generalized validation approach, called **cross validation**, consists in the repetition of such process several times, each time building different training, testing and validation datasets. In practice, it involves: i) dividing the dataset in two parts, each one composed of N_1 and N_2 samples (with $N_1,N_2 >> 1$); ii) using the first dataset for training and the second for testing; iii) collecting the performance results; iv) repeating from (i) varying the way the datasets are obtained; and v) analyzing the performance results obtained over the iterations, by comparing them, calculating the average or variance, and so on. In this work, we propose the use of **k-fold cross validation**, dividing the dataset in *k* folds, each one containing an equal number of samples $|N|/k$. At each step, *k-1* folds are used as the training dataset, while one fold is used as testing dataset, with the left-out fold being different each of the *k* times. The value of *k* can be chosen in an incremental manner, analyzing the behavior of the prediction model over each value. A widely used value in literature is $k=10$ (Dietterich 2002), which may be taken into account at a preliminary stage of the analysis of the prediction results.

3.5 Phase 4: Failure data accuracy analysis

The final step of the approach consists of analyzing the quality of the generated failure data, which is given by their similarity to data that would be collected when real failures occur. Several metrics have been proposed so far to analyze the quality of data used in information management systems. In this context, we believe that the concept of accuracy is sufficient for analyzing the similarity between synthetic and real failure data. In fact, given the possibility of representing the real workload in a multidimensional space composed of known characteristics, *accuracy* can be seen as the measure of the distance between a different set of data (e.g., the synthetic set) and the target data.

Obviously, a **necessary condition** for the synthetic failure data to be accurate is that the faults injected are realistic, as non-representative faults could lead to failures that *would not* occur in a real scenario (thus being not accurate). This is assured by the injection of realistic software faults using the Generic-Software Fault Injection Technique (G-SWFIT (J. A. Duraes and Madeira 2006)), as discussed in Section 3.2.2. However, injecting realistic faults is **not sufficient** to guarantee that the generated failure data are realistic, as the faults are not real, and the generation is also influenced by several others factors, including the workload used during the generation (see Section 3.3), whose impact on accuracy has to be studied.

Our proposal is to conduct a quantitative assessment of the accuracy of synthetic failure data, thus providing some degree of confidence as a sufficient condition for the use of the data in the context of failure prediction. The leading idea of such quantitative assessment is that generated (or synthetic) failure data can be considered accurate if there is a positive correlation between that data and real failure data, i.e., the failing behavior of the system due to fault injection is similar to a real failing scenario (for each single type of failure). In practice, we foresee two types of analysis that can be performed: **direct (or strong) accuracy analysis**, which directly correlates a synthetic dataset with a real dataset, and **indirect (or weak) accuracy analysis** that consists in comparing the performance of a failure prediction model trained with generated data and trained with real data, thus being more focused on the accuracy of failure data when used for the prediction task. In fact, while measuring the correlation between data can give interesting insights about the failure patterns in the real and the synthetic data, such method may suffer from the problem of being too much focused on data, thus not taking into account the impact of that data on the prediction quality of the trained models. This is why we foresee the need for the indirect accuracy analysis, as it is based on the prediction performance of the models trained with the synthetic failure data.

The problem is that assessing accuracy with data collected during real system operation is mostly not possible, as real data is usually not available. Our proposal is thus to **estimate the accuracy** of the generated failure data by partitioning the available synthetic data and by applying several estimation metrics to the resulting data subsets. In practice, we propose two quantitative estimation approaches:

1) **Direct or strong accuracy estimation** making use of metrics for computing the correlation between two sets of synthetic failure data, namely a *reference failure dataset DS_R* and a *validation dataset DS_V*, which must be obtained independently. Obtaining independent datasets can be achieved by injecting faults in different system's modules, thus emulating the potential diverse fault activation that could take place in a real scenario). The *direct* or *strong failure data correlation metrics* are identified by the symbol $\hat{\rho}$, and the leading hypothesis is that the closer the *failure data correlation* $\hat{\rho}(DS_R, DS_V)$ value is to one, the more the generated failure data are likely to be accurate.

2) **Indirect or weak accuracy estimation**, *indirectly* validating synthetic failure data accuracy by using metrics that portray the *performance degradation* of *prediction models* when varying the dataset. In practice, the performance values are obtained by training and testing the predictor with a reference dataset *DS_R*, and validating the prediction performance using a second and independent *validation dataset DS_V*. Such metrics can be referred to as *synthetization error*, and are identified by the symbol $\hat{\varepsilon}$. The closer the *synthetization error* is to zero, the more the generated failure data are likely to be accurate.

It is worth noting that the definitions presented must be applied to a single scenario, defined by a failure mode $\boldsymbol{\mathcal{F}}$ and a workload $\boldsymbol{W}$.

Although several metrics can be used, we propose a set of estimation metrics on the basis of empirical experience, with the aim of addressing the accuracy estimation problem. In practice, a deeper study on the definition or choice of optimal failure data accuracy estimation metrics is needed, but is considered as future work.

3.5.1 Direct data accuracy estimation

As strong accuracy estimation metric we propose the use of the Pearson's correlation for time series. The reason behind such choice stands in the fact that failure datasets are composed by time series (both in the case of numerical variables or categorical information, which should be converted to numerical elements, as discussed in Section 3.2.4), and the Pearson's correlation coefficient is a widely used method for measure the correlation between time series. The Pearson's correlation coefficient ρ between a *reference dataset DS*R and a *validation dataset DS*V is defined as the ratio between the covariance σ_{DS_R,DS_v} and the product of their standard deviations σ_{DS_R} and σ_{DS_v}, as follows[9]:

$$\rho_{DS_R,DS_v} = \frac{\sigma_{DS_R,DS_v}}{\sigma_{DS_R}\,\sigma_{DS_v}} = \frac{cov(DS_R, DS_v)}{\sigma_{DS_R}\,\sigma_{DS_v}} \tag{3.3}$$

with $-1 < \rho_{DS_R,DS_v} < 1$. The datasets are directly correlated if $\rho_{DS_R,DS_v} > 0$, and inversely correlated if $\rho_{DS_R,DS_v} < 0$, while the datasets are uncorrelated if $\rho_{DS_R,DS_v} = 0$. An optimal accuracy estimation would thus be $\boldsymbol{\rho_{DS_R,DS_v} = 1}$.

Another failure data correlation metric is proposed and presented in Chapter 4 for assessing the impact of using virtualization as a sandboxing solution for injecting software faults to overcome the limitation of using fault injection in production systems. As we will see, the goal is to analyze the similarity of failure data generated using a real system and several virtualized systems, defining a correlation metric to be applied to the data coming from the original target system and from its virtualized copies, based on the concept of failure symptoms, which are particular behaviors showed by one or more of the monitored variables.

[9] It must be noted that the Pearson correlation coefficient for discrete time series (as in the case of failure datasets) is usually identified by the letter r. However, in this section we present the correlation coefficient as it is used for continuous time series, i.e., ρ, as it is easier to analyze and comment.

3.5.2 Indirect accuracy estimation

We propose three metrics for characterizing the synthetization error ε (weak accuracy estimation) measuring the *performance degradation* of *prediction models* when varying the dataset. Although more metrics can be defined, we here present two metrics widely used as estimators (the relative error and the mean squared error), and a third one focusing on elements ordering, which can help in gaining confidence in the measures obtained by the estimators, and be an estimator itself. Each measure can be used with a single or several predictors, and two or more datasets. The proposed metrics are:

1) **Relative error** (*one predictor, two datasets*): $\hat{\varepsilon}$ is calculated as the relative error between performance measures (e.g., Prediction, Recall, ROC-AUC – see Chapter 2) obtained by *training and testing* a predictor with a *reference dataset DS$_R$* (divided into a training dataset *TDS$_R$* and a testing dataset *TTDS$_R$*) and *validating* it using a second and independent dataset *DS$_V$* (consisting of a testing dataset *TTDS$_V$* only). We denote as *reference performance* the performance obtained by using the reference dataset, while the *validation performance* is obtained by using the independent dataset.

$$\hat{\varepsilon} = \left|\frac{\mathcal{P}_{reference,DSR} - \mathcal{P}_{validation,DSV}}{\mu(\mathcal{P})}\right| = \left|\frac{\mathcal{P}_{ref,DSR} - \mathcal{P}_{val,DSV}}{\mathcal{P}_{ref,DSR}}\right| \tag{3.4}$$

In particular, $\mathcal{P}$ is the performance measure and $\mu(\mathcal{P})$ is the mean performance value of the predictor, which can be considered as the performance $P_{ref,DSR}$ obtained by using the reference dataset *DS$_R$*. It is worth noting that the relative error is measured by using the reference and validation datasets.

2) **Mean squared error, MSE** *(one predictor, two or more datasets)*: $\hat{\varepsilon}$ is calculated as the mean of the squared errors between the performance measures (e.g., Prediction, Recall, ROC-AUC) by training and testing a predictor with a *reference dataset DS$_R$* (training data *TDS$_R$*, testing *TTDS$_R$*) and *validating* it using one or more independent dataset *DS$_i$* (consisting of testing data *TTDS$_i$* only). The *validation performance* relative to different datasets is measure as:

(3.5)
$$\hat{\varepsilon} = MSE(\hat{\mathcal{P}}) = \frac{1}{n}\sum_{i=1}^{N}\left(\mathcal{P}_{reference,DS_R} - \mathcal{P}_{validation,DS_i}\right)^2 = \\ = \frac{1}{n}\sum_{i=1}^{N}\left(\hat{\mathcal{P}}_{DS_R} - \hat{\mathcal{P}}_{DS_i}\right)^2$$

where $\mathcal{P}_{DS_R}$ is the performance achieved by training and testing the predictor with *DS_R*, while $\mathcal{P}_{DS_i}$ is the performance obtained by training the predictor with *DSR* and testing/validating it with *DSi*. The advantage of computing $\hat{\varepsilon}$ as the MSE stands in the fact that it provides a better estimation, as we can use several validation datasets, and in the fact that MSE can be seen as a risk function, whose value can be interpreted as the risk of using the specific predictor under analysis trained with the generated dataset *DSR*.

3) **Kendall's tau distance** *(two or more predictors, two datasets)*: if two or more failure prediction systems are installed on a single target system, testing and validating the predictors with different datasets may result in different rankings, according to a given performance metric. In such scenario, if the rankings do not change when varying datasets then one can have more confidence on the performance values of each predictor. Hence, $\hat{\varepsilon}$ is calculated as the Kendall's tau distance by computing the pairwise disagreements between two ranking lists, the first consisting of the performance of several predictors trained and tested using a single reference dataset *DSR*, and the second obtained by validating their performance using one different dataset *DSV*. In particular, this distance is calculated as:

(3.6)
$$K(\tau_1,\tau_2) = \sum_{(i,j)\epsilon P} \bar{K}_{i,j}(\tau_1,\tau_2)$$
$$P = set\ of\ unordered\ pairs\ of\ distinct\ elements\ \tau_1\ and\ \tau_2$$
$$\bar{K}_{i,j}(\tau_1,\tau_2) = 0\ if\ i\ and\ j\ are\ in\ the\ same\ order\ in\ \tau_1\ and\ \tau_2$$
$$\bar{K}_{i,j}(\tau_1,\tau_2) = 1\ if\ i\ and\ j\ are\ in\ the\ opposite\ order\ in\ \tau_1\ and\ \tau_2$$
$$n\ is\ the\ length\ of\ the\ lists\ \tau_1\ and\ \tau_2$$
$$(\min disagreement) 0 \le K(\tau_1,\tau_2) \le \frac{n(n-1)}{2}\ (\max disagr.)$$

We can then define the normalized Kendall's tau distance as:

(3.7)
$$\bar{\bar{K}}(\tau_1,\tau_2) = 2\frac{\sum_{(i,j)\epsilon P}\bar{K}_{i,j}(\tau_1,\tau_2)}{n(n-1)}\ normalized\ distance$$

with:

$$(\min disagreement)\ 0 \leq \overline{\overline{K}}(\tau_1, \tau_2) \leq 1\ (\max disagr.) \quad \text{or} \quad (\max disagreement)\ 0 \leq 1 - \overline{\overline{K}}(\tau_1, \tau_2) \leq 1\ (\min disagr.) \tag{3.8}$$

The leading idea of using the Kendall's tau distance to estimate the accuracy of synthetic failure data stands in the fact that the effects on several independent failure prediction models must be the same (minimum disagreement). Also, the Kendall's tau distance can be used together with the other measures defined previously, when several different failure prediction models and several independent datasets are available. In fact, if the prediction models are *not* ordered in the same way over different datasets then the used datasets may be inducing some variation, which reduces the confidence in the accuracy of generated failure data, resulting also in a reduction of other weak accuracy estimations (e.g., the MSE or the relative error).

3.6 Case Study: The impact of the time dimension in failure prediction

To demonstrate the effectiveness of the proposed approach, in this case study we make use of synthetic failure data to **assess the performance of a prediction model** that implements a novel technique for improving the failure prediction task (Ivano Irrera, Pereira, and Vieira 2013). In practice, the proposed technique endows a prediction model to take into account the fact that data are made of time series, being thus trained considering the *temporal order* of the collected *data samples*.

The motivation for such technique grounds in the fact that systems overlooking the temporal ordering of sequential data can actually suffer from poor prediction performance (Dietterich 2002), as existing machine learning systems using time series or sequences of events build prediction models based on the hypothesis that data samples are *independent* (e.g., classifiers), resulting in a loss of information[10]. The prediction model used for the analysis is a Support Vector Machine (Cortes and Vapnik 1995) and the technique for including the time dimension in the prediciton task is called *Sliding Window*.

[10] Models explicitly taking into account the temporal ordering of data, as for instance regression-based models (see Section 2.2.2), are an exception.

3.6.1 The Sliding Window technique

Taking a predictor trained with a given dataset (as defined in Section 3.4), the output of a predictor *y(t)* at time *t* is dependent on the input vectors (or variables) *v(t)* relative to time *t.* Applying a sliding window to the dataset corresponds to making the output of the predictor *y(t)* dependent on *w* past time instants ($w \geq 1$), namely *(v(t), v(t-1), v(t-2), …, v(t-w-1))*. Therefore, a predictor trained with *windowed data* performs predictions taking into account such ordering.

As defined in Section 3.4, a dataset is composed of *n* different variables $\underline{v} = <v_1, v_2, \ldots, v_n>$, and each sample is labeled according to the failure prediction model presented in Section 3.2.5. In particular, given *t* as the time instant, $v_i(t)$ is a real number corresponding to the value of variable v_i at the time *t*, a *dataset sample* is a tuple:

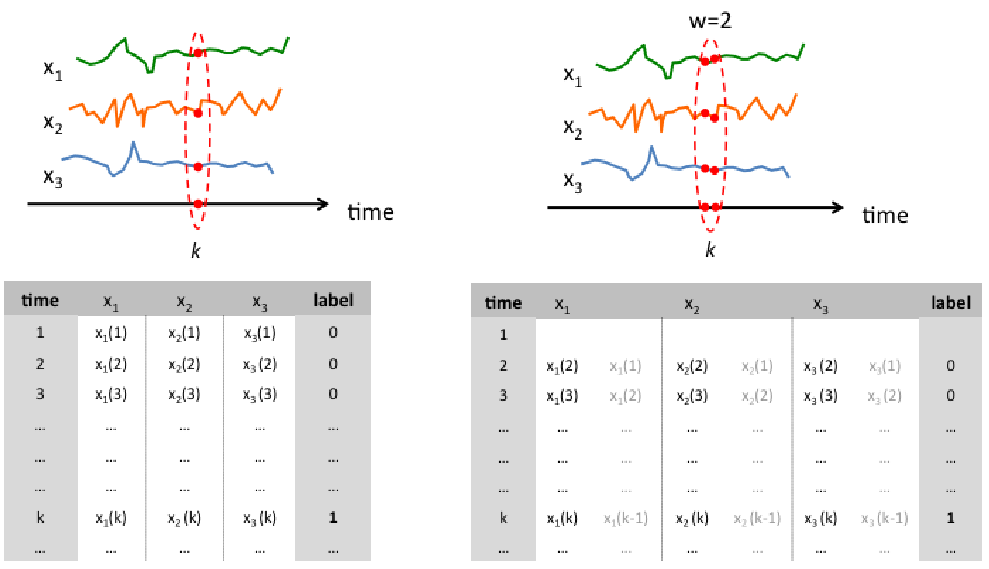
$$<v_1(t), v_2(t), \ldots, v_n(t), \mathbf{l(t)}> \tag{3.9}$$

with an associated label $l(t) \epsilon \{0,1\}$. Applying a *sliding window* of width $w>1$ to the dataset sample means that the label *l(t)* will depend on $w>1$ values of each variable v_i, in particular the value at time *t* and *w-1* past values:

$$\begin{array}{l} <v_1(t), v_1(t-1), v_1(t-2), ..., v_1(t-w), \\ \ v_2(t), v_2(t-1), v_2(t-2), ..., v_2(t-w), \\ \ \ldots, \\ \ v_n(t), v_n(t-1), v_n(t-2), ..., v_n(t-w), \mathbf{l(t)}> \end{array} \tag{3.10}$$

Therefore, the sliding window transforms the *data sample* from a vector of *n* components into a vector with *w*n* components, to which a label is associated. Note that the labeling does not change when using a sliding window, i.e., *l(t)* refers to the most recent time instant *t*. In fact, the failure event to predict at time *t* depends on the more recent values of the variables (relative to time *t*) and past values (in this case, values relative to the times *t-1, t-2, …, t-w*).

Considering a dataset composed as shown in equation (3.10), and the first time instant identified as *t=1* and a window *w*, the first time instant relative to the transformed dataset is *t=w*. In fact, if the first data sample is *t=1* then data samples relative to *t=0, t=-1, t=-2, …, t=-w+1* are not defined. Figure 3.7 shows an example of applying the sliding window technique to a dataset made of three features x_1, x_2 and x_3, using a window with width *w*=2. In particular, Figure 3.7 (b) shows that the technique cannot be applied to the time instant t=1, as $x_i(t-1)|_{t=1}=x_i(1-1)=x_i(0)$ are not defined. Hence, the datasets obtained by applying the sliding window technique start from the time instant *w* (in the example, t=2).



time	x_1	x_2	x_3	label
1	$x_1(1)$	$x_2(1)$	$x_3(1)$	0
2	$x_1(2)$	$x_2(2)$	$x_3(2)$	0
3	$x_1(3)$	$x_2(3)$	$x_3(3)$	0
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...
k	$x_1(k)$	$x_2(k)$	$x_3(k)$	**1**
...	...	...	...	...

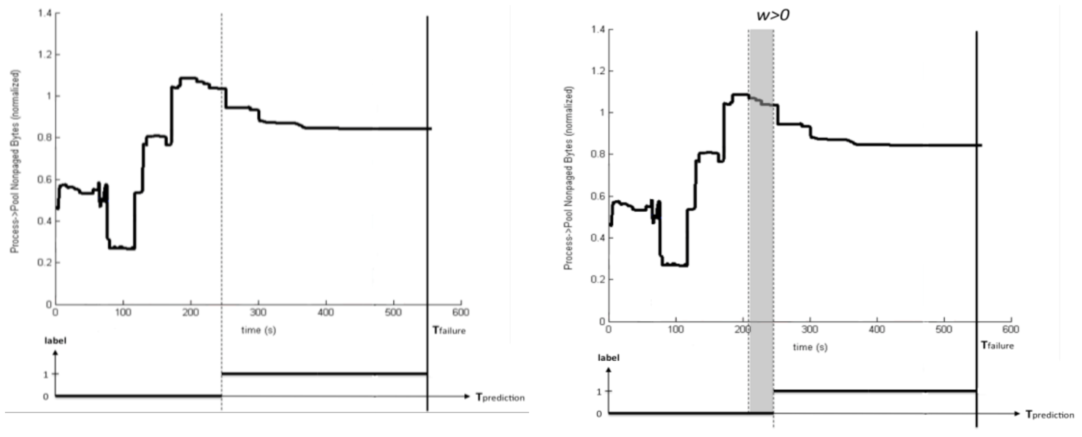
time	x_1		x_2		x_3		label
1							
2	$x_1(2)$	$x_1(1)$	$x_2(2)$	$x_2(1)$	$x_3(2)$	$x_3(1)$	0
3	$x_1(3)$	$x_1(2)$	$x_2(3)$	$x_2(2)$	$x_3(3)$	$x_3(2)$	0
...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...
k	$x_1(k)$	$x_1(k-1)$	$x_2(k)$	$x_2(k-1)$	$x_3(k)$	$x_3(k-1)$	**1**
...	...	...	...	...	...	...	...

(a) (b)

Figure 3.7 – Example of application of a sliding window with width w=2.

Figure 3.8 helps visualizing how the temporal ordering of data samples influences the failure prediction task. The label relative to the time instant *t*, originally dependent on the value of the variable *v* (*pool of non-paged bytes*, in the figure) at time *t* only (Figure 3.8 (a), relative to *w=1* time instant), is then related to *w* past variable values (Figure 3.8 (b), relative to *w*>1 time instants), which does not change the value of the label *l(t)*.

It is important to emphasize that the sliding window must be applied both to *training datasets* and to *testing datasets*, as the prediction task must be applied to the same data features.



(a) w=1 **(b) w>1**

Figure 3.8 –The temporal ordering of data in failure prediction.

3.6.2 Definitions and set-up

The first step of the data generation process includes identifying the failures to be predicted, building failure detectors, selecting the variables to characterize the behavior of the system (that will compose the datasets), and defining the workloads to be used and the faults to be injected. Note that, although the choices regarding the workload and the failure modes considered (among others aspects) in the present case study may not fit many real world scenarios, they do serve for demonstrating the effectiveness of the proposed approach.

The case study is based on an environment that includes a Windows XP SP3 machine (the *target system*), installed on a virtual machine running on top of a VMWare vSphere server. A *controller* machine is in charge of controlling the experiments and analyzing the failure data coming from the system. The configurations of the machines is as follows:

1) **Machine #1 (target)**: Intel i5-650@3.60GHz machine, 8GB RAM, running a Windows XP OS (SP3) in a VMWare vSphere server based on ESXi v5.0. Running the target system as a virtual machine on a VMWare vSphere server (Frappier 2014) gave us the possibility of saving the state of the system at the beginning of the fault injection campaign, and restoring that saved state at the end of each run. This check-pointing functionality copies the configuration of the virtual machine, as well as the data contained in the virtualized storage disk and its running state (e.g., state and data of the processes in execution, values contained in the CPU registries, etc.).

2) **Machine #2 (controller)**: Intel i5-650@3.60GHz machine, 8GB RAM, running a Windows XP OS (SP3), used to: *(a)* control the experiments (start/stop the experiments), *(b)* remotely command and control the fault injection tool, *(c)* force the reboot of the machines in case of failures (including in hanging situations), *(d)* collect the data and store them in a Microsoft SQL Server 2008, and *(e)* analyze data.

The **libSVM** libraries implement the SVM predictor (Chang and Lin 2011) on top of which we implemented the sliding window by wrapping the libSVM libraries using MATLAB scripting to feed the SVM with data windowed according to the approach presented in the previous section.

Regarding the **failures to predict**, we empirically focused on *crashes* and *hangs*, which are two failure modes frequently observed in the Windows XP OS (“Support - Windows Help” 2015). Making a correspondence to the C.R.A.S.H. scale (Koopman et al. 1997), a system crash corresponds to the OS becoming corrupted and the machine crashes or reboots, while a system hang corresponds to the OS becoming unresponsive and needing to be terminated by force.

A **failure detector** able to detect the occurrence of the two failure modes mentioned above was implemented. In practice, the detector continuously monitors the target system to detect failures in the following way:

1) a crash is detected when the system does not respond to a *ping* for a certain time T_{max_ping}. The failure time T_F is obtained by considering the first time instant in which the system became unresponsive;
2) a hang is detected if the target system responds to a *ping*, but it hangs on executing a given set of operations. Again, the failure time T_F is obtained by considering the first time instant in which the system became unresponsive, identified by the time instant when the first not executed operations were sent to the system.

The target system runs **two workloads**, namely the WinRAR application (WKL_1) (RAR Lab), compressing a file using the RAR algorithm with the low compression option, and the COSBI OpenSourceMark computer benchmarking suite (WKL_2) ("COSBI OpenSourceMark"), a more complex workload that includes computation and input/output intensive tests, compression algorithms, disk and memory accesses, etc. (we consider that these workloads include generic operations that computer systems perform frequently, being thus adequate for the present case study). The combination <Workload, Failure mode> allows defining four different **scenarios** for the analysis: <WKL_1, Crash>, <WKL_1, Hang>, <WKL_2, Crash>, and <WKL_2, Hang>.

We adopted a Windows-based **software fault injection tool** implemented at University of Coimbra following the G-SWFIT recommendations (J. A. Duraes and Madeira 2006) for the fault injection task. Such tool is able to inject software faults at machine-code level both in binary files and in running processes (user-mode only). However, due to the fact that the Windows OS includes a protection for avoiding certain system files from being changed, the fault injector was limited to inject software faults in running processes of the operating system, but faults were injected before starting the collection of data, thus simulating residual faults from the perspective of the data collection process.

The **faultload** is based on the fault types defined in (J. A. Duraes and Madeira 2006) and previously presented in Section 3.2.2. Based on previous experience, we focused the fault injection on the code of the **svchost.exe** process and of the linked **dynamic library kernel32.dll** (containing functions for handling the OS memory usage), which are key resources of the Windows XP OS (e-Testing Labs 2001; Kalakech et al. 2015; Kalyanakrishnam, Kalbarczyk, and Iyer 1999; "Support - Windows Help" 2015) The fault injection tool was able to automatically generate thousands of *code mutants* by analyzing the fault locations matching a specific pattern depending on the type of software fault, being each fault identified by the tuple *<fault type, fault location, code mutant>*. In order to design a feasible experiment, a subset of the faults was selected based on the relevance of their *locations* (details on the number of faults injected and their impact are presented in the next subsection). For this, we used a profiling tool (*Luke Stackwalker*), which helped identifying the functions and modules executed along several runs of the workloads considered. As previously discussed, the selection of the most executed modules of the target system does not invalidate the representativeness of the injected software faults.

Regarding the **variables to monitor**, we considered at set of variables reflecting the state of the operating system and the usage of the hardware resources, as the symptoms of the failures considered may manifest at the OS and at lower levels (e.g., an increase in the number of context switches/s). In practice, we monitored 233 numerical variables, at the sample rate of one value per second, using the Logman tool that is included in Windows OSs family, and afterwards conducted a three-step feature selection to reduce the number of variables.

In the first step we eliminated from the 233 monitored variables the ones that have a constant or *null* value in all the runs. In the second step the variables were correlated using a classic linear correlation metric (Pearson correlation coefficient), filtering out variables having a correlation greater than 0.9 between each other. In the third step a classical wrapper approach with backward elimination was applied to the set of variables that resulted from the previous step (feature selection). A SVM was used to validate the selection, taking its ROC-AUC as reference for characterizing the quality of the variable set and a k-fold cross validation (k=5) for avoiding results biasing. The resulting set consists of 25 variables (out of the initial 233) for each scenario <WKL, Failure> and for each couple of values (Δt_I, Δt_P) considered in our analysis.

An excerpt of 10 out of the 25 variables for <WKL$_1$, Crash> and (Δt_I, Δt_P)=(10s, 5s) is shown in Table 3.2.

Table 3.2 - Selected variables, for <WKL#1, Crash> and (Δt_I, Δt_P)=(10s, 5s) (excerpt)

Variable ID	Variable name	Monitored component
123	Pool Nonpaged Allocs	Memory
117	Page Faults/sec	Memory
201	C2 Transitions/sec	Processor
209	Exception Dispatches/sec	System
220	System Calls/sec	System
156	Current Disk Queue Length	PhysicalDisk
94	Avg. Disk sec/Transfer	LogicalDisk
139	Semaphores	Objects
182	Pool Nonpaged Bytes	Process
39	Sync Data Maps/sec	Cache

3.6.3 Data generation, dataset building and failure predictor training

Each workload was executed 3500 times (500 GR + 3000 FIR), with a maximum execution time of T_{MAX}=180s each. Table 3.3 summarizes the OS failures observed during the fault injection campaign. The failure data was collected in less than one month, with a total of 195 failures (121 when running WKL_1 and 74 when running WKL_2), being OS hang the most frequent type of failure.

In each fault injection run, a single fault was injected approximately 70 seconds after starting the execution of the workload (this value was defined based on the analysis of the ramp up time of the tested configurations). If a failure occurs within the time T_{MAX} (with a 10% time tolerance to consider potential delays due to the system scheduling), the run is labeled as a *Failure Run*, otherwise the run is labeled as a *Non-Failure Run* (both are Fault Injection Runs).

As shown in Table 3.3, failures were observed in a small subset of the fault injection runs. This is expectable, as there is no guarantee that the locations chosen for the injection are actually executed (J. A. Duraes and Madeira 2006). We must highlight the fact that the failure occurrence is in average 2%, which is similar to the activation rate obtained by other authors that used the G-SWFIT technique to inject faults in different systems (e.g., (Roberto Natella et al. 2013)). This gives us some confidence on the representativeness of the types of faults injected.

The failure prediction parameters used in this case study were Δt_l=[20s, 50s] and Δt_p=[5s, 25s], chosen according to the workload execution time. In this context we used several values for Δt_l and Δt_p, while fixed Δt_e=5000 samples, a value that keeps low the time for training a prediction model (found experimentally). The warning interval Δt_w, that can be identified based on the performance of the failure predictors, is not considered in our case study for the sake of simplicity.

The failure data generated was used to train the SVM prediction model, using several values for the sliding window, namely w={2, 3, 4, 7, 10} seconds. The training of the SVM failure prediction model comprised the optimization of a set of parameters, among which the most important are γ (kernel function parameter) and C (cost of allowing training errors) (Cortes and Vapnik 1995). The parameters optimization was performed in two steps, making use of the ROC-AUC prediction

Table 3.3 - Failures generated

Workload	**# Golden Runs**	**# Fault Injection Runs**	**Failures detected**		
			Total %	***System Crash %***	***System Hang %***
WKL_1	500	3000	121 *(4.03%)*	46 *(1.53%)*	75 *(2.5%)*
WKL_2	500	3000	74 *(2.47%)*	6 *(0.2%)*	68 *(2.27%)*

Table 3.4 - The parameters of the analysis

Parameter	Values
Failure Modes	Crashes, Hangs
Workloads	WKL_1, WKL_2
Predictor	SVM (Gaussian kernel)
Variable selection	Backward elimination + wrapper approach
Predictor Optimization	Grid search (gross) + Deepest descend (fine)
(γ, C) (Grid search)	γ= [2^{-10},1], C=[2^{-1}, 2^7]
Δt_l (Failure prediction)	20, 30, 40, 50 s
Δt_p (Failure prediction)	5, 10, 15, 20, 25 s
Window size (w)	2,3,4,7,10 s
Results validation	5-folds cross validation

performance metric (see Section 2.2.3), and applying the k-fold cross validation with k=5 for validating the generalization of the prediction results. In the first step, we performed a grid search on intervals of values that we experimentally defined for each SVM hyper-parameter. Then the optimal (γ, C) values found were used as a starting value for a fine-search (second step), which consisted in solving a non-linear minimization problem through the use of a gradient descend method. Table 3.4 summarizes the parameters of the prediction analysis, as discussed before.

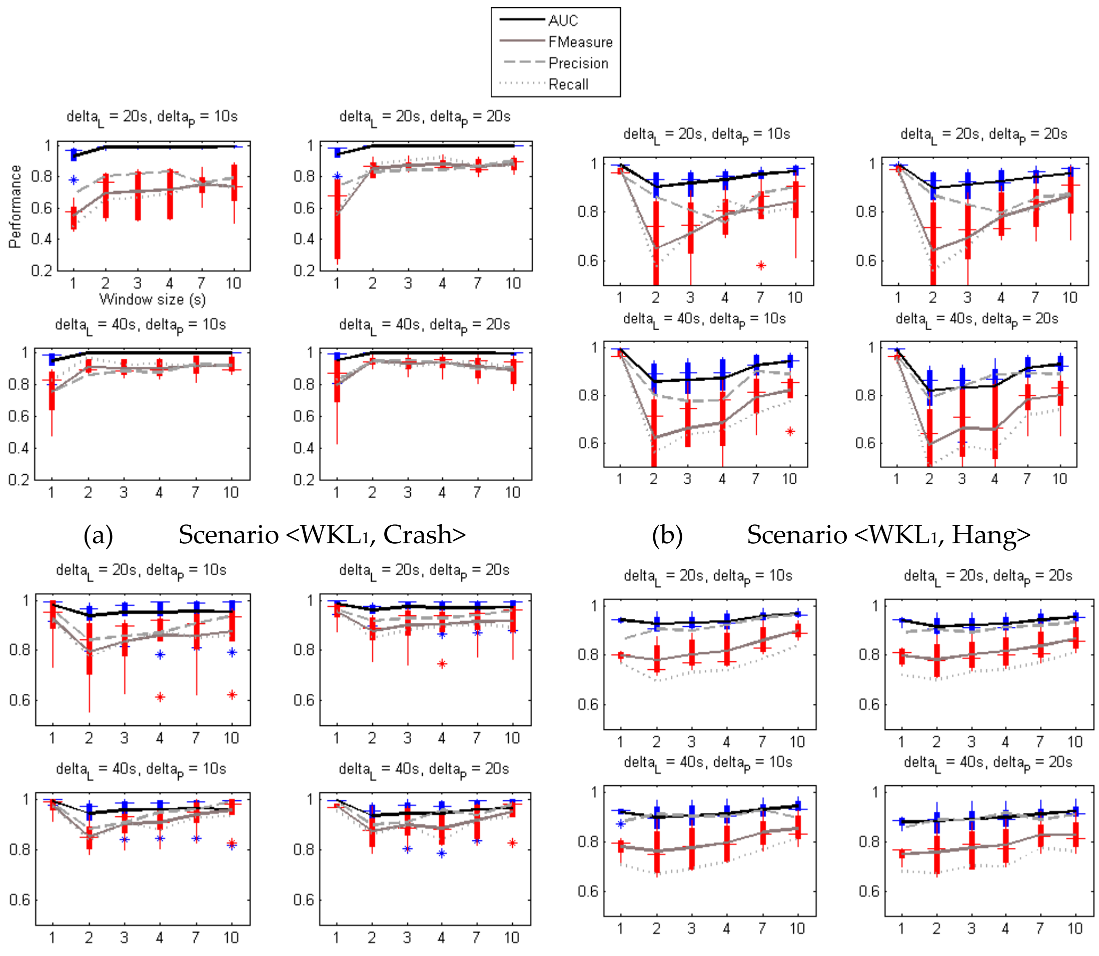
3.6.4 Results and discussion

The performance of the predictor is measured in terms of ROC-AUC, F-Measure, Precision and Recall (see Section 2.2.3). The relative cost of using windowing in terms of the predictor training time (excluding the feature selection and parameters optimization time) is analyzed too. Please note that the optimal threshold of the SVM, relative to its ROC cut-off point, is used to compute the F-Measure, Precision and Recall measures (for details about prediction thresholds, see Section 2.2.3). The optimal threshold depends on the scenario, the values (Δt_l, Δt_p), the features selected, and the parameter *w*.

Figure 3.9 shows the performance of the SVM classifier for each scenario <*WKL, Failure*>, without the application of the sliding window (*w*=*1*) and also using windows of 2, 3, 4, 7, and 10 seconds. Each subplot in the figure represents the results for each couple *(Δt_l, Δt_p)*, using box-and-whiskers diagrams, representing the minimum, lower quartile, median, upper quartile and maximum performance values obtained over the different folds.

The use of the time dimension improves the prediction in scenario <WKL_1, Crash> (Figure 3.9 (a)), both in terms of the ROC-AUC and the F-Measure, for any couple of values *(Δt_l, Δt_p)*. In addition, the windowing also seems to reduce the variation of the ROC-AUC (revealed by the cross validation). In scenario <WKL_2, Hang> (Figure 3.9

(b)) the performance also improves, but only for a sliding window larger than 4 seconds. This may be due to the fact that *t-4* brings more information to the prediction than the time instants *t-1*, *t-2* and *t-3*, or to the need to tuning the prediction parameters *(Δt_l, Δt_p)*. In the remaining scenarios <WKL2, Crash> and <WKL1, Hang> windowing does not improve the performance along *w*. This is an acceptable behavior, as it gets harder for the SVM to find the optimal hyper-plane to correctly classify the data. In fact, the dimension of the features grows from 25 to *w**25 (up to 250 features, in our case). Nonetheless, in all the scenarios it can be observed that the performance when using the sliding window tends to increase, as the window size *w* gets larger. This is due to additional information given to the classifier, but more scenarios need to be explored in order to confirm such trend.



(a) Scenario <WKL1, Crash> (b) Scenario <WKL1, Hang>

(c) Scenario <WKL2, Crash> (d) Scenario <WKL2, Hang>

Figure 3.9 - The impact of windowing on ROC-AUC

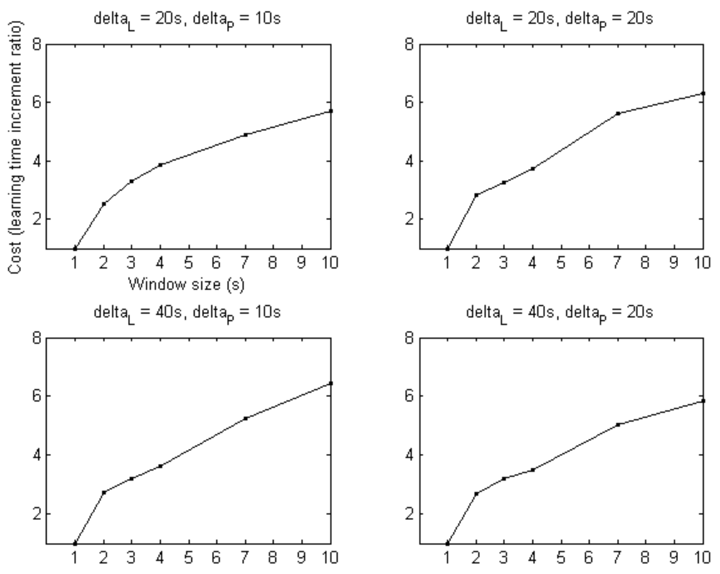


Figure 3.10 - The learning time increment ratio <WKL2, Hang>

Taking as example the results in Figure 3.9 (b) and (d) (thus fixing on workload WKL2), we can see that the sliding window improves the predictor performance in case of Hangs (d), while generally degrading the prediction when trying to predict Crashes (b). Moreover, comparing the cases (a) with (b) and (c) with (d) in Figure 3.9, we can note that the prediction performance is generally higher when predicting Crash failures. This last analysis highlights the fact that the performance of a predictor depends on many factors, including the nature of the failures to be predicted and the workload.

Figure 3.10 shows an insight on the growth of the computational cost, i.e., the time needed for training the SVM, with respect to the case without windowing. The growth is logarithmic with the number of features, and the training time can reach an increase of 6 times. The logarithmic growth seems consistent with the SMO (Sequential Minimization Optimization Method) method used in libSVM (Chang and Lin 2011). Nevertheless, the use of a sliding window seems a viable approach to improve the prediction of software failures.

3.6.5 Accuracy analysis

In this case study we estimated the accuracy of the synthetic failure data using the **weak accuracy estimation** analysis, as we are interested on understanding the impact that the generated data has on the performance of the failure prediction (such estimation metrics can give some confidence on the use of failure data to train failure predictors for working in a real operational scenario).

For sake of simplicity, we considered only the following configurations of the prediction model:

- *FPA_1*, SVM classifier with no sliding window (w=1s)
- *FPA_2*, SVM classifier with w=2s
- *FPA_3*, SVM classifier with w=3s
- *FPA_4*, SVM classifier with w=4s

For performing the analysis, we used the dataset from the previous section as the reference dataset (here referred to as **DS_1**), and generated a second dataset (validation dataset **DS_2**) by injecting faults in a different system module used by the svchost.exe process, namely **ntdll.dll** that contains the OS kernel functions. The settings for the dataset generation are similar to the ones used for DS_1, in particular in what concerns the faultload that included the same fault types and the same number of faults for each type. We computed and analyzed the ROC-AUC performance measure of the prediction model for the four scenarios <*WKL, Failure Mode*>, considering a fixed (Δt_l, Δt_p)=(30s, 15s), and conducted a 5-fold cross validation.

Table 3.5 shows the performance results (ROC-AUC) for the scenarios <WKL_1, Crash> and <WKL_2, Crash>, while Table 3.6 shows the results for <WKL_1, Hang> and <WKL_2, Hang>. Based on such results, we estimated the **mean squared error MSE*** and the **relative performance error ε*** of each predictor. In practice, we computed such estimation measures based on the mean of the performance measure μ (thus ε_μ^* and MSE_μ^*) and their median value M (thus ε_M^* and MSE_M^*) applied to the results obtained from the 5-fold cross validation. The gray tones in the tables are related to the predictor's ROC-AUC performance, with higher performances associated to darker gray tones[11].

[11] Note that results in the tables show that the predictors that achieved the highest performance were FPA_2 and FPA_4, again confirming that the use of the sliding window can improve the quality of failure prediction.

Table 3.5 - ROC-AUC and synthetization error (a, crash)

(a) Crash								
	WKL$_1$				***WKL$_2$***			
	FPA$_1$	***FPA$_2$***	***FPA$_3$***	***FPA$_4$***	***FPA$_1$***	***FPA$_2$***	***FPA$_3$***	***FPA$_4$***
Performance DS$_1$	0,6852	0,8137	0,6713	0,9024	0,9391	0,9977	0,9461	0,9878
	0,9427	0,9719	0,9560	0,9909	0,9778	0,9998	0,9859	0,8919
	0,7374	0,9909	0,7283	0,8975	0,9331	0,9756	0,9430	0,7958
	0,9518	0,9708	0,9652	0,9965	0,9356	0,9996	0,9576	0,9893
	0,7425	0,9898	0,7375	0,9897	0,7985	0,9979	0,8691	0,8854
μ(Perf.)	*0,8119*	*0,9474*	*0,8117*	*0,9554*	*0,9168*	*0,9941*	*0,9403*	*0,9100*
M(Perf.)	*0,7425*	*0,9719*	*0,7375*	*0,9897*	*0,9356*	*0,9979*	*0,9461*	*0,8919*
Performance DS$_2$	0,9223	0,9841	0,9073	0,9719	0,9341	0,9867	0,9432	0,9878
	0,9662	0,9137	0,9847	0,9953	0,9778	0,9978	0,8691	0,9919
	0,5665	0,9262	0,5284	0,9971	0,8433	0,9756	0,9461	0,9358
	0,8501	0,9797	0,8482	0,9953	0,9354	0,8979	0,9076	0,9893
	0,3649	0,9876	0,6675	0,9884	0,7885	0,9981	0,9891	0,9854
μ(Perf.)	*0,7340*	*0,9583*	*0,7872*	*0,9896*	*0,8958*	*0,9712*	*0,9280*	*0,9780*
M(Perf.)	*0,8501*	*0,9797*	*0,8482*	*0,9953*	*0,9341*	*0,9867*	*0,9432*	*0,9878*
Weak accuracy estimation								
$\mathbf{1-\bar{\bar{K}}_{\mu}}$	**0,833** (1 - 1/6)				**0,5** (1 - 3/6)			
MSE$_\mu$\*	**0,039**	**0,005**	**0,012**	**0,017**	**0,011**	**0,012**	**0,006**	**0,034**
ε$_\mu$\*	**0,106**	**0,011**	**0,031**	**0,035**	**0,023**	**0,024**	**0,013**	**0,070**
$\mathbf{1-\bar{\bar{K}}_{M}}$	**1** (1 - 0)				**0,333** (1 - 4/6)			
MSE$_M$*	**0,054**	**0,004**	**0,055**	**0,003**	**0,001**	**0,006**	**0,002**	**0,048**
ε$_M$*	**0,127**	**0,008**	**0,131**	**0,006**	**0,002**	**0,011**	**0,003**	**0,097**

Table 3.6 - ROC-AUC and synthetization error (b, hang)

	(b) Hang							
	WKL_1				***WKL_2***			
	FPA_1	***FPA_2***	***FPA_3***	***FPA_4***	***FPA_1***	***FPA_2***	***FPA_3***	***FPA_4***
Performance DS_1	0,7226	0,9881	0,8060	0,9147	0,8541	0,9943	0,8914	0,9768
	0,6999	0,8557	0,6529	0,7987	0,6826	0,9953	0,6925	0,9533
	0,9369	0,9967	0,9400	0,9972	0,6573	0,9889	0,6902	0,9502
	0,8926	0,9886	0,9329	0,9590	0,6356	0,8598	0,6203	0,8255
	0,8823	0,9959	0,8703	0,9639	0,7069	0,9943	0,7178	0,8794
μ(Perf.)	*0,8269*	*0,9650*	*0,8404*	*0,9267*	*0,7073*	*0,9665*	*0,7224*	*0,9170*
M(Perf.)	*0,8823*	*0,9886*	*0,8703*	*0,9590*	*0,6826*	*0,9943*	*0,6925*	*0,9502*
Performance DS_2	0,7934	0,5158	0,8389	0,7993	0,9391	0,8414	0,8066	0,8923
	0,5829	0,9629	0,6454	0,8659	0,9778	0,9787	0,7318	0,8753
	0,7740	0,9645	0,7750	0,8926	0,9331	0,9211	0,8303	0,8988
	0,9282	0,9967	0,9403	0,9879	0,9356	0,9875	0,7538	0,8976
	0,6901	0,6773	0,6875	0,7594	0,7985	0,9385	0,7455	0,7571
μ(Perf.)	*0,7537*	*0,8234*	*0,7774*	*0,8610*	*0,9168*	*0,9334*	*0,7736*	*0,8642*
M(Perf.)	*0,7740*	*0,9629*	*0,7750*	*0,8659*	*0,9356*	*0,9385*	*0,7538*	*0,8923*
	Weak accuracy estimation							
$1-\overline{\overline{K}}_{\mu}$	**0,833** (1 - 1/6)				**0,666** (1 - 2/6)			
MSE_{μ}*	**0,037**	**0,071**	**0,032**	**0,033**	**0,105**	**0,017**	**0,026**	**0,026**
ε_{μ}*	**0,097**	**0,172**	**0,081**	**0,076**	**0,229**	**0,035**	**0,066**	**0,061**
$1-\overline{\overline{K}}_{M}$	**0,833** (1 - 1/6)				**0,666** (1 - 2/6)			
MSE_{M}*	**0,054**	**0,013**	**0,048**	**0,047**	**0,127**	**0,028**	**0,031**	**0,029**
ε_{M}*	**0,140**	**0,027**	**0,123**	**0,108**	**0,271**	**0,060**	**0,081**	**0,065**

Considering the four scenarios, we can see that the mean squared error MSE_μ* is between 0.5% and 3%, with outliers 7% and 10%, for the cases (FPA_2, WKL_1, Hang) and (FPA_1, WKL_2, Hang) (corresponding to the use of an SVM without sliding window, or with a very small window with, also confirming the results from the previous section). The values of MSE_M* (thus relative to the median ROC-AUC) are very similar to the ones calculated on the mean, varying between 0.4% and 4%. However, we observed that the outlier MSE_μ*=10% corresponds to MSE_M*=12%, as one of the ROC-AUC values in that case (FPA_2, WKL_1, Hang) is 0.5158. Although it may seem better to use the MSE_M* and the median of the values, this choice may hide situations in which a failure prediction performs very poorly. On the other hand, the estimation of the relative performance error ε_μ* and ε_M* showed values between 1% and 10% in terms of ROC-AUC, with two outliers of about 17% (corresponding to MSE_μ*=10% by (FPA_2, WKL_1, Hang)) and 22% (corresponding to MSE_μ*=10% by (FPA_1, WKL_2, Hang)).

A closer look to the results shows that the proposed metrics recognize a similarity between the ROC-AUC performance using DS_1 and DS_2, which may reflect a similarity of the datasets DS_1 and DS_2. Moreover, the behavior of both MSE and ε is similar in all the analyzed scenarios.

By analyzing the *normalized Kendall's tau distance* $\overline{\overline{K}}_\mu$ (relative to ε_μ* values) and $\overline{\overline{K}}_M$ (relative to ε_M* values) we can observe that both the average and median performance values lead to a variation in the order of the ranking of the FPAs of at most one inversion in the WKL_1 scenario (e.g., in the Crash case (a), from <FPA_2, FPA_3, FPA_1, FPA_4> to <FPA_4, FPA_2, FPA_3, FPA_1>, according to the average of the performance values μ only), while between two and four inversions in the WKL_2 scenario (e.g., in the Crash case (a), from <FPA_2, FPA_3, FPA_1, FPA_4> to <FPA_4, FPA_2, FPA_3, FPA_1>). This means that the data generated using WKL_1 causes a *lower variation* in the models' prediction performance (between 0.5% and 4% using the MSE_μ*), suggesting a *good accuracy*. On the other hand, for WKL_2 the Kendall's tau distance reflects a quite high variation of the prediction models, suggesting a *low accuracy*, a result that is confirmed by the values of MSE and ε, as shown before.

The results obtained allow us to conclude that **the proposed approach is able to generate accurate failure data**, however the workload used may have non-negligible effects on the failure data accuracy. Moreover, the joint use of the proposed metrics allows different degrees of confidence for the failure data generated in the different scenarios. In particular, MSE* and the ε* can be considered valid metrics for estimating the accuracy of synthetic failure prediction data, while the *tau distance* provides a validation of such metrics, giving a confidence to the estimation values proportional to $1 - \overline{\overline{K}}$.

3.7 Final remarks

This chapter presented a practical approach for generating failure data for training and testing failure prediction models on concrete system installations, based on the use of software fault injection. The idea is that injecting faults on a particular system increases the probability of the system to fail, thus accelerating the collection of data, and that failure prediction models can be trained using those data. The fault injection technique adopted is the G-SWFIT technique that allows the emulation of residual software faults, a necessary condition for generating accurate failure data, i.e., data similar to data that would be collected from the same system due to the activation of an existing residual (real) fault.

The chapter presented guidelines for the implementation of the approach, which is based on four phases. The first is the definitions phase, in which one must define the faults to inject, the workload (real, realistic or synthetic) to be executed during the fault injection campaign, as well as the variables to be collected from the target system and the characteristics to take into account when predicting failures. The second phase is the failure data collection, in which one must implement an experimental setup to collect data. The third phase defines the rules for organizing the collected data into datasets to be used for training and testing failure prediction models. The fourth and last phase presents the guidelines for performing the analysis of the accuracy of the generated failure data.

We demonstrated the effectiveness of the approach by analyzing the results obtained in a case study in which an SVM-based failure prediction model was used to predict failures in a Windows XP OS system, running on a virtual machine hosted by a VMWare vSphere hypervisor. In particular, we assess the performance of a technique for improving the prediction performance that takes into account the temporal ordering of the data samples (using a *sliding window*). We were able to cause 191 failures (crashes and hangs), running two different workloads (a simple and a more complex one) 3500 times each. Results show that the use of a sliding window is a viable approach to improve the prediction of software failures. Moreover, we found that different workloads and failure modes influence the performance results. Finally, the case study demonstrated that we were able to train, test and study the behavior of the failure prediction model in four different scenarios (considering two different failure modes occurring when using two different workloads) in a short time interval.

The chapter also presented an estimation of the accuracy of the generated synthetic failure data using weak accuracy estimation metrics based on the performance results in four cases. Results show that the prediction performance of models trained with data generated using the simplest workload present a low variation when tested on independent datasets, which is an indication of *good accuracy*. On the other hand, the more complex workload seems to cause a higher variation of the prediction models, thus suggesting a *low accuracy* of the failure data generated in that scenario. Such result shows that the proposed approach can be used to generate accurate failure data and that the joint use of the proposed metrics allows getting some degree of confidence on the failure data generated in different scenarios. On the other side, results also showed a margin for the improvement of the quality of generated failure data, which can be achieved by using the proposed measures to study the conditions under which more accurate failure data can be generated.

It must also be observed that the simpler environment provided by the use of virtualization in the case study allowed a much faster data collection, as the system could be restored after the injection of software faults. However, an analysis of the impact of virtualization technology on the generation of failure data is needed in order to better understand the potential of use of such technology. This is precisely the goal of Chapter 4.

Chapter 4
Virtualization as a support for the generation of failure data

Although the solution proposed in Chapter 3 should be applied while running the target system, injecting faults in a real system has the obvious drawback of causing unacceptable failures during operation. A potential solution is to generate the failure data in an environment that mimics the behavior of the target system and in which software faults can be safely injected. However, although the use of an alternative machine/environment can be a valid solution, it is not feasible in many situations (e.g., due to the additional costs of buying new hardware, the difficulty of restoring the target system's state during the experiments, etc.).

In this chapter we propose and assess the use of **virtualization** as a sandboxing solution for supporting the process of generating failure-related data by injecting faults in a virtualized copy of the system. Virtualization technology is a solution that provides an easily updatable and disposable environment, where faults can be safely injected and from where failure data can be collected, while allowing running multiple copies of a system (thus reducing the time needed for the failure data generation). In addition to this, the virtualization solutions available nowadays (see (Chiueh and Brook 2005)) also offer features like saving and restoring the system state by using native check-pointing and restoring functionalities, thus facilitating the removal of the injected faults, as already shown in the case study presented in Chapter 3.

A great number of commercial systems are nowadays based on virtualized environments (e.g., virtualized servers in server consolidation frameworks, enterprise virtualization solutions providing user virtualized resources, clouds). In fact, in the last decade there was a growing tendency for software systems to be run in virtualized environments, as these permit the reduction of the cost associated with buying new hardware, the reallocation of resources (e.g., storage, computational power), and fault tolerance based on the distribution of nodes (e.g., (Fu 2009; Nagarajan et al. 2007; Polze, Troger, and Salfner 2011; Boyd and Dasgupta 2002; Machida, Kawato, and Maeno 2010)). In particular, many systems are built on top of cloud technology, which is mainly based on virtualization (Rimal, Choi, and Lumb 2009), including secure and dependable systems (Bessani et al. 2013; Gurumurthy et al. 2015; Jain and Singh 2014; Magalhaes and Moura Silva 2013). Moreover, several works addressing the problem of failure prediction in cloud systems are present in literature (Chen, Lu, and Pattabiraman 2014; Gunawi et al. 2011; Otsuka et al. 2014; Sonoda et al. 2012; Yukihiro Watanabe and Matsumoto 2014). For these reasons, we believe that using virtualized environments for generating failure data is a solution that should be studied, particularly in what regards:

- **The impact of the virtualization environment on the generated failure data**: failure data collected from a virtualized environment may significantly differ from data generated on the original system, as data depend on the behavior of the underlying software and hardware system's layers. Besides leading to the same types of failures, it is important that the faults injected produce *similar failure data patterns* that can equivalently be consumed by failure prediction systems;
- **Solutions for replicating the target system into the virtualization environment**: several solutions may be adopted (e.g., replicating the OS in the virtualized system, use middleware solutions). As this is a too vast problem, it is out of the scope of this work. Thus, we assume that the original target system *can be* virtualized (i.e., the system can be replicated in a virtualized environment) or that it is a virtualized system itself. We believe that such assumption is acceptable for this first study on the impact of using virtualization to generate failure data by fault injection.

We propose a solution for computing the similarity of failure data generated on a virtualized copy of a target system with failure data generated on the original system. Such approach should be used when adopting virtualization as a sandboxing solution for the generation of such data. The approach is based on the identification of failure symptoms (i.e., patterns that a set of data show before a failure occurs (Felix Salfner, Lenk, and Malek 2010)) presented by the monitored variables, their correlation with the failures observed in both the original and virtualized systems, and the comparison of the correlation values (of the individual variables) gathered from the different environments by using statistical testing methods. In practice, the approach proposed enables studying the impact of the virtualization layer in order to understand the possibility of using virtualization for generating synthetic failure data *adequate* for training and testing the failure prediction systems that will run on the original target system (i.e., *is the data generated in a virtualized copy of a system adequate for training failure prediction mechanisms to be run in the original system?*). It is worth noting that, although focused on answering such research question, the proposed approach can be also considered a solution for the **direct estimation of failure data accuracy** problem discussed in Chapter 3.

The chapter is organized as follows. Section 4.1 overviews the approach for assessing the similarity of failure data. Sections 4.2, 4.3, 4.4 and 4.5 present the phases of the approach. In particular, Section 4.2 describes the generation of failure data by using the failure generation approach proposed in Chapter 3. Section 4.3 describes the approach for identifying failure symptoms. Section 4.4 describes the correlation of symptoms with the failures observed, and Section 4.5 describes the assessment of the failure data similarity, based on the use of statistical tests. A case study is presented in Section 4.6, in which we analyze the effectiveness of our approach on failure data collected from a Windows XP OS environment running on a physical machine and on several virtualized environments. Finally, Section 4.7 concludes the chapter.

4.1 Overview of the approach

To assess the similarity between failure data gathered from a virtualized copy of a system and data gathered from the original target system, we propose a four-step approach based on the concept of failure symptoms:

1) **Generating failure data** in the original target system and in the virtualized system, using the approach for failure data generation based on realistic software fault injection presented in Chapter 3. Both systems must run under the *same conditions*, i.e., using the same *workload*, the same failure model, and the same *faultload*.
2) **Identifying failure symptoms** by analyzing the monitored variables (from both the original and the virtualized copy). The proposal is to use an *anomaly detection-based method* for building a *normal behavior profile* for each variable using data collected during Golden Runs, and then comparing the variable's behavior during Failure Runs and other Golden Runs against that profile. A variable presents a symptom if its behavior during a FIR differs from its normal behavior profile, according to a given measure. The reasoning for such approach is that the simplest way for detecting symptoms is by identifying deviations from the nominal behavior. Although more complex methods are possible, their use is not central to the present work.
3) **Correlating failure symptoms with the failures** observed. In particular, a variable is said *correlated* to a failure if its values present some symptom when the failure occurs and do not present any symptom during Golden Runs. We model such situations based on the contingency table (see Section

2.2.3), with a True Positive corresponding to a variable presenting a symptom when a failure is observed, and a True Negative indicating that the variable shows no symptoms during a Golden Run. On the other hand, a variable can be *uncorrelated*. In practice, we quantify the correlation of each individual variable with respect to all the failures observed on a given system. The correlation value of each variable with the observed failures is validated using a k-cross validation approach (for details, Section 3.4).

4) **Assessing the accuracy** (or similarity) of the failure data gathered from the two different systems, by comparing the correlation values of the individual variables by means of *statistical analysis*.

We must highlight the fact that the proposed approach can also be used in the case both systems are virtualized or non-virtualized (e.g., the comparison between virtualized systems can be useful when one must migrate from a technology to another, or to assess the impact that an upgrade on a virtualization technology can have on the generated failure data).

Our approach can be formalized as follows: let M_i be the *i-th* machine (virtual or not) with $i=1, 2, \ldots, |M|$, hosting a system S running a workload WKL_k, and let $frd(M_i)$ be the *failure related data* coming from M_i, with associated failure events $\underline{F}$ (failures) and $\underline{G}$ (no failures, events occurring during Golden Runs). Data are made of a set of numerical variables $v_i(t)$, $i=1, 2, \ldots, |V|$, where V is the group of variables monitored from a single machine, each variable describing one particular aspect of the system (e.g., mutexes, allocated memory, threads running). Each variable $v_i(t)$ may present a **failure symptom** relative to a failure event f_k (true symptom, or True Positive), **no symptom** relative to no-failure event g_k (true no-symptom, True Negative), or may present a symptom when no failure actually occurred (false symptom, or False Positive) or no symptom when a failure occurred (false no-symptom, or False Negative). A variable $v_i(t)$ presents a (positive) **correlation** C with the entire set of occurred failures $\underline{F}$ (and the set of no-failure events $\underline{G}$), if the value of C is proportional to the number of true symptoms observed when a failure occurred (or no-symptoms when no failure occurred), and inversely proportional to the number of false symptoms when a failure *does not* occur (false no-symptoms when a failure *does* occur).

Considering the quantity $C(frd(M), \underline{F}, \underline{G})$[12] (or just $C(frd(M), \underline{F})$) as the measure of the correlation of the failure data (for the collection of variables v_i) coming from machine M with the failures occurred on such machine, we say that the failure data coming from machines M_i and M_j, with $i \neq j$, are similar when the correlation of each failure dataset with the failures occurred $C(frd(M_i), \underline{F})$ and $C(frd(M_j), \underline{F})$, have similar values.

[12] It is worth to note that the quantity $C(frd(M_j), \underline{F}, \underline{G})$ is actually a vector, as the data are made of several variables. Nonetheless, we use such form for the sake of simplicity.

The correlation measure should be considered variable-wise, thus the quantities C(*frd*(M), $\underline{F}$) of each variable $v_i(t)$, C_{vi}(*frd*(M_i), $\underline{F}$), must have similar values on different machines (variable-wise similarity). Hence, we say that sets of failure data coming from different machines are similar (general similarity) if:

$$(4.1) \quad \begin{aligned} & S = \sum_{v=v_1}^{v_N} S_v\big(C_v(frd(M_i,F)),\ C_v(frd(M_j,F))\big) > \lambda \\ & S_v = \{0,1\} \\ & S \epsilon \mathbb{R} \end{aligned}$$

where S is a measure of the general similarity, S_v is a measure of the variable-wise similarity, and λ is an acceptance value for the different sets of failure data to be considered similar (ideally equal to the number of variables |V|).

The correlation measures (thus also the similarity measures) are validated making use of the *k-fold cross validation* technique (Box, Hunter, and Hunter 2005) to ensure that there is no dependence on a particular dataset. In practice, *k* different folds from each *frd*(M_i) are built according to the approach presented in Section 3.4. Thus, the similarity measures and the correlation values of each variable monitored from the machine M_i are *vectors* of *k* components $\underline{C}_{vi}$(*frd*(M_i)) (Box, Hunter, and Hunter 2005). For verifying the hypothesis that failure related data coming from machines M_i and M_j (with *i≠j*) are *similar*, the variable-wise correlation vectors $\underline{C}_{vi}$(*frd*(M_i)) and $\underline{C}_{vi}$(*frd*(M_j)) must also have similar values. This condition can be verified by statistical testing, aiming at rejecting the hypothesis that $\underline{C}_{vi}$(*frd*(M_i)) is similar to $\underline{C}_{vi}$(*frd*(M_j)). In practice, equation (4.1) becomes:

$$(4.2) \quad \begin{aligned} & S = \sum_{v=v_1}^{v_N} S_v \begin{pmatrix} C_v^1(frd(M_i)), C_v^2(frd(M_i)), \ldots, C_v^k(frd(M_i)) \\ C_v^1\left(frd(M_j)\right),\ C_v^2\left(frd(M_j)\right), \ldots,\ C_v^k\left(frd(M_j)\right) \end{pmatrix} > \lambda \\ & C_v^k\ identifying\ the\ C_v\ value\ relative\ to\ a\ single\ fold\ k \\ & S_v = \{0,1\} \\ & S \epsilon \mathbb{R} \end{aligned}$$

where, for each variable *v* (or v_i), the correlation values relative to each fold *k*, $\underline{C}^k_{vi}$(*frd*(M_i)), are compared between two machines M_i and M_j. In practice, we can state that **virtualization has low impact on the data generation process if the datasets from M_i and M_j are similar** (according to equation (4.2)).

4.2 Phase 1: Failure data generation

The first phase is the generation of failure data by applying the approach proposed before. We thus refer to Chapter 3 for details on such approach, presenting here only the formalization of the generated data.

The organization here used for the datasets differs from the one in Chapter 3 in what concerns the labeling, as presented in Figure 4.1. In particular, each variable v_i from a Fault Injection Run FIR_k is associated to a Failure or a No-Failure event instead of being associated to a set of labels. Therefore, the Golden Data is made of variables collected when no failure occurred, thus associated to No-Failure events, while Failure Data is made of variables collected when failures where observed.

	v_1	**v_2**	**v_3**		**v_n**	**Failure/ No failure event**
	$v_1(1)$	$v_2(1)$	$v_3(1)$	…	$v_n(1)$	
	$v_1(2)$	$v_2(2)$	$v_3(2)$	…	$v_n(2)$	
	$v_1(3)$	$v_2(3)$	$v_3(3)$	…	$v_n(3)$	
	$v_1(4)$	$v_2(4)$	$v_3(4)$	…	$v_n(4)$	
	…	…	…	…	…	
FIR_k	$v_1(k)$	$v_2(k)$	$v_3(k)$	…	$v_n(k)$	***Failure/ No failure***
	$v_1(k+1)$	$v_2(k+1)$	$v_3(k+1)$	…	$v_n(k+1)$	
	$v_1(k+2)$	$v_2(k+2)$	$v_3(k+2)$	…	$v_n(k+2)$	
	$v_1(k+3)$	$v_2(k+3)$	$v_3(k+3)$	…	$v_n(k+3)$	
	…	…	…	…	…	
	$v_1(T_F)$	$v_2(T_F)$	$v_3(T_F)$	…	$v_n(T_F)$	

(a)

	v_1	**v_2**	**v_3**		**v_n**	**Failure/ No failure event**
	$v_1(GR_1)$	$v_2(GR_1)$	$v_3(GR_1)$	…	$v_n(GR_1)$	*No failure*
	$v_1(GR_2)$	$v_2(GR_2)$	$v_3(GR_2)$	…	$v_n(GR_2)$	*No failure*
Golden Data	$v_1(GR_3)$	$v_2(GR_3)$	$v_3(GR_3)$	…	$v_n(GR_3)$	*No failure*
	…	…	…	…	…	…
	$v_1(GR_G)$	$v_2(GR_G)$	$v_3(GR_G)$	…	$v_n(GR_G)$	*No failure*

						F/NF
	$v_1(FIR_{f1})$	$v_2(FIR_{f1})$	$v_3(FIR_{f1})$	…	$v_n(FIR_{f1})$	*Failure*
Failure Data	$v_1(FIR_{f2})$	$v_2(FIR_{f2})$	$v_3(FIR_{f2})$	…	$v_n(FIR_{f2})$	*Failure*
	$v_1(FIR_{f3})$	$v_2(FIR_{f3})$	$v_3(FIR_{f3})$	…	$v_n(FIR_{f3})$	*Failure*
	…	…	…	…	…	…
	$v_1(FIR_{fF})$	$v_2(FIR_{fF})$	$v_3(FIR_{fF})$	…	$v_n(FIR_{fF})$	*Failure*

(b)

Figure 4.1 - Data from a single Failure Run i (a) and a complete (global) dataset (b)

The global dataset, including all the Golden and Failure Data, is here divided in **Training or Profiling Dataset**, made of a given percentage of Golden Data and used for creating the normal behavior profile for each variable, and **Testing or Analysis Dataset**, which includes the remaining Golden Data and Failure Data. In practice, a part of the golden runs is used for building the profile, and the remaining for later validating the correlation of the variable with the observed failures (i.e., the variable should present no symptom in a golden run). We propose a proportion of 25% for the Profiling Dataset and 75% for the Analysis Dataset, aiming at using more data for the failure symptoms identification, though such percentages can be easily tuned.

Note that our approach includes the use of k-fold cross validation with a run-by-run (or run-wise) partitioning of Golden and Failure Data, as in Chapter 3, which permits to have *k* distinct Profiling and Analysis datasets.

4.3 Phase 2: Symptoms identification

To compute the correlation *C* for each v_i over the set $\underline{F}$ in the Analysis Dataset (set of occurred failures, caused by fault injection), we first need to identify symptoms. In this section we propose a symptom recognition approach based on an anomaly detection method, which should be applied to the individual variables selected for failure prediction. In particular, we propose to build a **profile** $p(v_i)$ for each variable using Golden Data from the Profiling Dataset, modeling its normal normal behavior. The key idea is the following: if during a Failure Run the values of the variable v_i fall *out-of-profile*, a symptom is identified.

The profile $p(v_i)$ is created by computing the maximum and the minimum values obtained in the runs where no faults are injected (and no failures are observed), thus $p(v_i)=p(v_i(t),g_j)$, $j=h_1,h_2,\ldots,h_r$, considering only part of the golden runs. An example of a profile is shown in Figure 4.2. The profile or model is represented by two curves, namely the **upper and lower bounds** of the values of the variable seen along the golden runs (including a tolerance value for each bound, which can be set empirically and should take into account the variations that the values of a variable

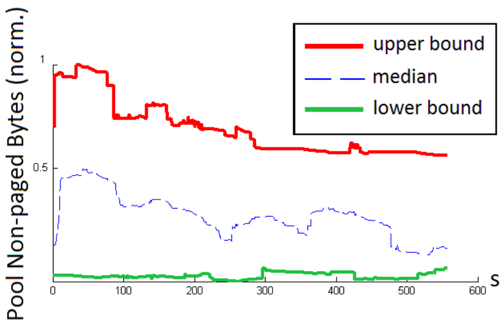


Figure 4.2 – The profile of the "Pool of Non-Paged Bytes" variable (normalized)

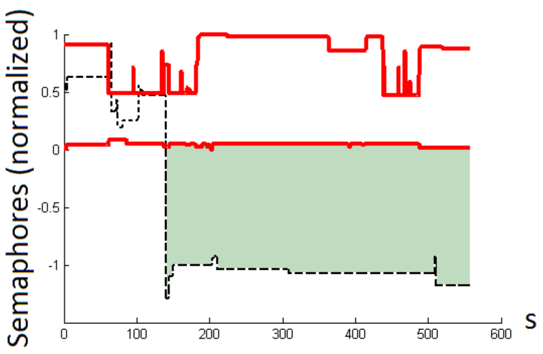


Figure 4.3 – A symptom identified on the "Semaphores" variable (normalized)

can present along runs). The central curve represents the median values of the variable (just for presentation purposes).

A variable v_i, monitored during a given failure run *r*, presents a *symptom* if, compared to the profile $p(v_i)$, the *area* between the bounds and the deviating values is greater than a threshold T_{vi} (relative to each variable v_i). In the example in Figure 4.3, a software fault is injected at *t=68* seconds after starting the workload execution (*t=0*), and the variable values overrun the bounds at *t=135* seconds, showing a reduction in the number of semaphores the operating system is managing, probably caused by a part of the operating system that stopped working.

Let [x,y] be the group of points in which the parameter shows overrunning values, $p(v_i)$ be the profile of the variable v_i, and T_{vi} a threshold value for deciding if the parameter v_i presents a symptom. We state that a variable v_i presents a symptom $Symptom(v_i, FIR_i)$ in the *i-th* Fault Injection Run if the area between the variable values from the *i-th* FIR $v_i(t, FIR_i)$ and the maximum and the minimum bounds expected (defined by $p(v_i)$) is greater than T_{vi}:

$$Symptom(v_i, FIR_i) = 1 \Leftrightarrow \int_x^y |v_i(t, FIR_i) - p(v_i(t))|\, dt \geq T_{vi} \tag{4.3}$$

$$\begin{aligned} &[x,y] = B_u \cup B_l \\ &B_u = \{t: v_i(t) \geq MAX(p(v_i(t)))\} \\ &B_l = \{t: v_i(t) \leq \min(p(v_i(t)))\} \end{aligned} \tag{4.4}$$

The threshold T_{vi} is a real value that must be obtained, for each variable, in a way that allows recognizing the maximum number of symptoms and minimizing the possible false negatives, using any kind of performance function. In this work, we propose an **adaptive approach for the definition of the threshold T_{vi} for a variable v_i,** making use of feedback information about the correlation that a variable would have when using a given threshold value. In practice, the threshold T_{vi} is defined as:

(4.5)
$$
\begin{aligned}
&T_{vi} \epsilon \mathbb{R} / \underset{T_{vi}>0}{\arg\max}\left(\boldsymbol{CorrelationMetric}(v_i, \boldsymbol{\mathcal{F}}, \boldsymbol{G})\right)\\
&\boldsymbol{\mathcal{F}} = \{failures\ F_k, with\ k\ \epsilon\ ]1, |FIR|]\}\\
&\boldsymbol{G} = \{no - failures\ G_h, with\ h\ \epsilon\ ]1, |GR|]\}
\end{aligned}
$$

As mentioned before, the *correlation metric* must be proportional to the number of true symptoms and inversely proportional to the number of false symptoms. As an example, equation (4.6) shows two metrics addressing the condition above, namely F-Measure and Prediction (for details, see Section 2.2.3).

E.g.,

(4.6)
$$
\begin{aligned}
&1)\ T_{vi} \epsilon \mathbb{R} / \underset{T_{vi}>0}{\arg\max}\left(F - Measure(v_i, \boldsymbol{\mathcal{F}}, \boldsymbol{G})\right)\\
&2)\ T_{vi} \epsilon \mathbb{R} / \underset{T_{vi}>0}{\arg\max}\left(Precision(v_i, \boldsymbol{\mathcal{F}}, \boldsymbol{G})\right)
\end{aligned}
$$

In practice, starting from a threshold value T^0_{vi}, the symptoms identification depends on the threshold value T^*_{vi} maximizing the correlation value C between the symptoms that the variable v_i shows and the failures observed in the system. The approach proposed is depicted in Figure 4.4. In particular the symptom identification phase and the correlation phase are organized in a feedback loop, which is solved by applying a maximization algorithm. We do not specify any particular optimization algorithm, but a simple approach is to adopt a time-limited grid search (i.e., a heuristic searching the best value in a defined interval), with a maximum time as the termination criterion for the search algorithm. Finally, the correlation value obtained using such schema is the maximum correlation value that

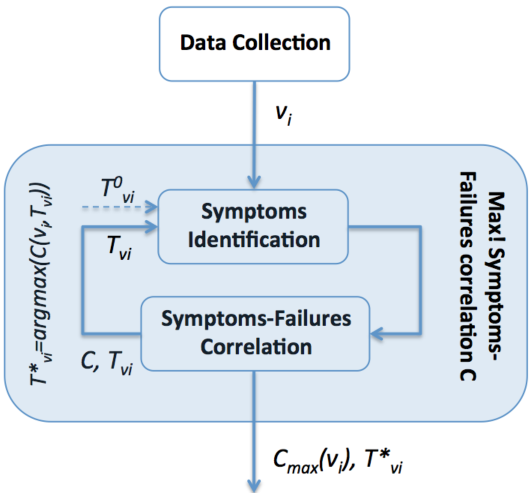

Data Collection
v_i
T^0_{vi}
T_{vi}
Symptoms Identification
Symptoms-Failures Correlation
C, T_{vi}
$T^*_{vi}=argmax(C(v_i, T_{vi}))$
Max! Symptoms-Failures correlation C
$C_{max}(v_i), T^*_{vi}$

Figure 4.4 – The adaptive schema for threshold definition, symptoms identification and symptoms/failures correlation

can be obtained varying the threshold. The correlation value associated to each variable v_i is the value corresponding to the optimal threshold, thus $C_{max}(v_i)$.

Obviously, the threshold may influence metrics like the number of false positives (i.e., number of times the surface metric exceeds the threshold during fault injection runs in which failures where not observed) and the coverage (i.e., number of times the surface metric exceeds the threshold during fault injection runs in which failures where observed) of each variable. This is why an adaptive threshold is needed, as it allows mitigating the cases where small noisy deviations lead the values of variables to go slightly out of the typical bounds (or go out of the bounds for a very short time frame).

4.4 Phase 3: Symptoms and failures correlation

In order to assess the similarity between different failure data, we define a *function C* for each variable v_i, measuring the *correlation* between the symptoms presented by the variable when failure (and no-failure) events occurred on a specific system. The correlation values must be in the range [0,1] (1 means that the variable values are highly correlated with the failure occurrence), and the correlation metric chosen should be proportional to the number of true symptoms and inversely proportional to the number of false symptoms. In this work we propose to correlate the variables' symptoms with the observed failure/no-failure events using the ROC-AUC measure, i.e., the Area under the Receiver Operating Characteristic curve (see Section 2.2.3) The reason that stands behind this choice is the fact that ROC analysis represents a solution for the adaptive approach presented in the previous section, where the optimal value $C_{max}(v_i)$ associated to each variable v_i is the ROC-AUC, and the optimal threshold T^*_{vi} is the ROC's cut-off point (i.e., the point corresponding to the optimal correlation value), considering each variable as a prediction model (see Figure 4.5).

In practice, a ROC curve is composed of points corresponding to the measures **true positives rate** (or Sensitivity) and **false positives rate** (1-Specificity), obtained according to the contingency table, while varying a decision threshold. The threshold used by the ROC analysis corresponds to the threshold T_{vi}, hence, varying such threshold and computing *tpr (true positive rate)* and *fpr (false positive rate)* allows obtaining a ROC curve. The true positives rate and false positives rate are defined as follows:

(4.7) *True Positives Rate / Sensitivity/ Recall*

$$\frac{TP}{TP+FN} = \frac{Symptoms\ corresponding\ to\ \boldsymbol{failures}}{All\ the\ occurred\ \boldsymbol{failures}}$$

(4.8) *False Positives Rate / 1-Specificity*

$$\frac{FP}{TN+FP} = \frac{\boldsymbol{False}\ symptoms}{All\ the\ \boldsymbol{no-failure}\ cases}$$

The more the true positives rate is near to 1 and the false positive rate is near to 0, the higher is the correlation between the variable's symptoms and the failures (such situation corresponds to an ideal correlation a ROC-AUC=1). An example of a ROC curve is presented in Figure 4.5. The black convex curve is the ROC curve relative to a correlated variable, with a 0.5 < ROC-AUC < 1 (as ROC-AUC=0.5 corresponds to an uncorrelated curve, for which the variation of the threshold does not cause any change in the couple (Sensitivity, 1-Specificity)). The ROC of a perfectly correlated variable is a single point corresponding to (Sensitivity, 1-Specificity)=(1,1) and has a ROC-AUC=1 (also here the variation of the threshold does not influence the correlation metrics). Moreover, an inversely correlated variable corresponds to a concave ROC curve. Finally, the cut-off point (defined only for convex curves) is the point corresponding to the optimal correlation measures, i.e., the higher true positives rate with the lower false positives rate.

The correlation values $C(v_i, \{F, G\})$ calculated for each variable v_i are finally validated using *k-fold cross validation*. This consists of partitioning the dataset coming from M_j into k_{M_j} folds, dividing the data "run by run", taking advantage of the fact that the dataset is already partitioned (in golden or failure *runs*). We take into account this existing division of data, without invalidating the results. In fact, the statistical properties coming from the use of *k-fold* cross validation are not altered, as such super-partition of the data is still a partition. Each fold is obtained by combining golden data and failure data, considering the runs of the associated events g_l and f_l, obtaining groups like $fold_r = \{GD(g_{r1}, g_{r2}, \dots g_{rr}), FD(f_{r1}, f_{r2}, \dots, f_{rr})\}$. The folding is done in such a way that each fold ends containing at least data related to one failure.

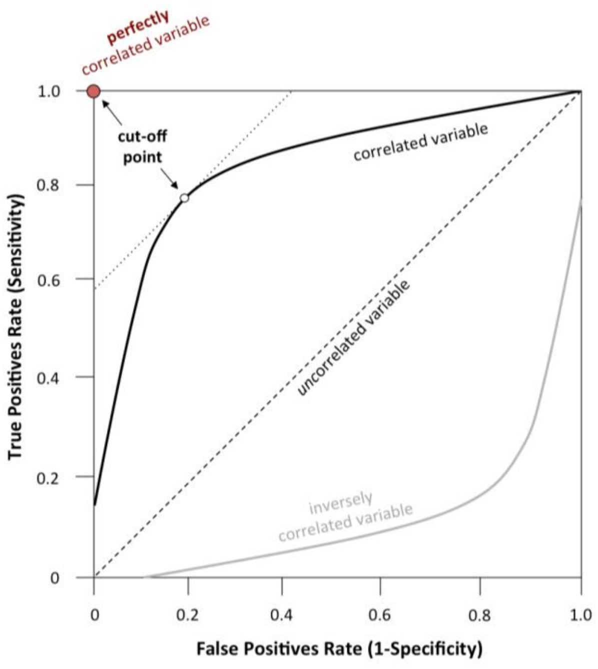


Figure 4.5 – ROC curves relative to single variables

As the number of failures observed in each machine may vary, the number of folds may also vary from one machine to another. This way, it may happen to have few folds on some machines, but it does not happen to have the same failures in different folds, neither to have folds with no failures. Finally, we compute the correlation values for each fold relative to the failures occurred on M_j: for each variable v_i we compute the correlation $\underline{C}(v_i)$ of length $k_{\mathrm{M}j}$, that is $\underline{C}(v_i)_{Mj}=(C_{p1}(v_i),\ C_{p2}(v_i),\ \ldots,\ C_{k\mathrm{M}j}(v_i))_{Mj}$.

4.5 Phase 4: Failure data similarity analysis

The last phase of the approach consists of testing the hypothesis that the samples $\underline{C}(v_i)_{M1}$ from M_1, $\underline{C}(v_i)_{M2}$ from M_2, etc., are similar. More specifically, we are interested in ***not* rejecting the *null hypothesis***, being:

> ***H_0***: *"the variable $v_i(t)$ presents similar failures correlation values on the two machines M_i and M_j"*,

also corresponding to:

> ***H_0'***: *"the samples $\underline{C}(v_i)_{Mj}$, with $j=1, 2, \ldots, |M|$, come from the same distribution"*,

which we assume to be true. If the test finds no evidence for rejecting the null hypothesis, we can continue considering that (at least in this case) virtualization has no influence on the generated failure data. On the other hand, if the test rejects the null hypothesis, then the data are not similar.

We compute the similarity of the *correlation statistic* (or *vector*) of each variable v_i monitored from the virtualized system and the original target system (or any kind of different machines, as mentioned previously) using the Kruskal-Wallis statistical test (Box, Hunter, and Hunter 2005) that is used for analyzing the variance of *N* distributions, as we verified that the correlation distributions are non-parametrical (or generic, as nothing can be said on the type of distributions to analyze).

4.6 Case study: Impact of virtualization in the generation of failure data

The case study presented in this section, aiming at assessing the impact of using virtualization environments for generating failure data, is based on the analysis of the data gathered from a set of five machines, one hosting the original target system (M_1) and the other four (M_2, M_3, M_4 and M_5) hosting virtualized copies of that system. The virtualization environments under study include both Type I (or Full

Virtualization) and Type II (or Hardware layer virtualization) hypervisors, thus considering a representative set of virtualization solutions (see Section 2.3).

The five monitored machines and the controller machine that compose the experimental setup have the following configurations:

- **Machine #1**: Intel i5-650@3.60GHz; 8GB RAM; Windows XP OS (SP3); no virtualization (hosts the original system).
- **Machines #2 and #3**: virtual machines hosted on Intel i5-650@3.60GHz systems with 8GB RAM, and running *Windows XP OS (SP3)*. Machine#2 runs on a Citrix XEN server v5.6.10, and Machine#3 runs on a VMWare vSphere server based on ESXi v5.0. These provide two virtual versions hosted on top of Type II Hypervisors, with an hardware configuration made of one out of the two cores of the hosting system's CPU (Intel i5-650@3.60GHz) and 1 GB RAM.
- **Machines #4 and #5**: virtual machines hosted on Intel P4 HT@3.00GHz systems with 2GB RAM, and running *Windows XP OS (SP3)*. Machine#4 runs on a Oracle's VirtualBox, and Machine#5 runs on a VMWare Player, both on top of Windows XP OSs. These provide two virtual versions hosted on top of Type I Hypervisors, with an hardware configuration made of the hosting system's CPU (Intel P4 HT@3.00GHz) and 1 GB RAM.
- **Machine #6 (Controller)**: Intel i5-650@3.60GHz; used to: *a)* control the experiments, *b)* remotely command and control the fault injection tool, *c)* force the reboot of the machines in case of failures, *d)* collect the data and store them in a Microsoft SQL Server 2008, and *e)* analyze the data using MATLAB.

As a real and a virtualized environment can have different hardware configurations (mainly in terms of CPU and RAM), in this case study the configurations of the defined virtual machines (Machines #2, #3, #4, #5) are different from the one of the real machine (Machine #1). On the other hand, all the virtual machines share similar configurations. With such setup, we can infer the impact of both the virtualization environment and the hardware configuration on the failure data. In particular:

- if the *real* and *all the virtual machines* share the same set of variables, it is likely that both the virtualization and the hardware do not impact on the generated failure data;
- if the *real* and *one virtual machine* share the same set of variables, it is likely that both the virtualization and/or the hardware do impact on the generated failure data;
- if the failure data from the *real* machine is *different* from the data collected from the virtualized environments, but the latter share among them similar

data, then it can be said that the virtualization is impacting on the generation of failure data;

- if the failure data from the *real* machine is *different* from the data collected from the virtualized environments, and the similarity depends on the virtualized machine (the sets of variables are different on the different hypervisors), then both the virtualization environment and the hardware do impact the generated failure data.

4.6.1 Data generation

We monitored 233 numerical variables representing the state of the operating system (OS) resources, at the sample rate of one value per second, using the Logman tool that is included in Windows OSs family.

The failure data are obtained by monitoring the systems while running two different workloads (one based on the WinRAR application (RAR Lab) and the other on the COSBI OpenSourceMark benchmark suite ("COSBI OpenSourceMark")) and targeting two distinct failure types: system Crash (OS becomes corrupted and the machine crashes or reboots) and system Hang (application or OS becomes unresponsive and must be terminated by force). The G-SWFIT tool was installed on each monitored machine (original and virtualized copies) and injected software faults in the OS to maximize the impact of faults on the system operations. The faultload (specifying which faults, where and when to inject) is exactly the same for all the machines, and was defined similarly to the case study presented in Section 3.6. We recall that to ensure a higher fault activation ratio, specific portions of the OS were previously selected as prime candidates for fault injection, resulting in the selection of the *kernel32.dll* and *ntdll.dll* system modules used by the system process *svchost.exe* (Generic Host Process for Win32 Services).

Table 4.1 presents the overall characterization of the experiments. A single fault was injected in each FIR approximately 70 seconds after starting the execution of the workload (defined based on the analysis of the ramp up time of the systems). Failures were observed in a small subset of the fault injection runs. This is due to the large number of possible fault locations, which reduces the probability of injecting a fault in a code location executed in a given experiment (J. A. Duraes and Madeira 2006). It is important to note that the failure occurrence rates have similar values on all the machines, ranging from 2% and 4% (considering both failure modes), which suggests that the fault activation is not strictly dependent on a particular system, although the differences among systems result in occurrence rate variations. Also, the faults injected caused more hang failures than crashes in all machines, which is in line with the fault injection results presented in the case study in Chapter 3. From a high-level observation, we can say that the Windows XP OS reacts in the same way

Table 4.1 - Failures generated

machines	wkl	# GR	# FIR	Failures detected		
				Total %	*Crash %*	*Hang %*
Real (M_1)	#1	1000	3000	*2.84% (80)*	0.37% (11)	2.3% (69)
	#2	1000	3000	*2.2% (66)*	0	2.2% (66)
Citrix XEN server (M_2)	#1	1000	3000	*4.3% (129)*	0.8% (24)	3.5% (105)
	#2	1000	3000	*2.2% (66)*	0	2.2% (66)
VMWare vSphere server (M_3)	#1	1000	3000	*4% (120)*	1.5% (45)	2.5% (75)
	#2	1000	3000	*2.9% (87)*	0.2% (6)	2.7% (81)
VMWare Player (M_4)	#1	1000	3000	*1.83% (55)*	0.13% (4)	1.7% (51)
	#2	1000	3000	*2.4% (72)*	0.5% (15)	1.9% (57)
Oracle VirtualBox (M_5)	#1	1000	3000	*1.47% (44)*	0.07% (2)	1.4% (42)
	#2	1000	3000	*2.3% (69)*	0.3% (9)	2% (60)

to software faults despite of the hardware used, which may anticipate the fact that data collected from different machines may share some similarities.

4.6.2 Symptoms similarity estimation

The comparison of the ROC-AUC correlation distributions for estimating the data similarity was performed using the Kruskal-Wallis test applied to each of the 233 variables, a generalization of the ANOVA test for non-parametric distributions (as we cannot make any assumptions about the distribution of the data). The significance level considered is α=0.05 and the sample values $\underline{C}(v_i)_{Mj}$ were obtained from each machine M_j using 20 folds.

Table 4.2 presents *examples* of the failure correlation distribution of the same variable collected on different machines, for guiding the reader in understanding the results obtained using the Kruskal-Wallis test. The table presents the ROC-AUC correlation values for the variables *Transition Faults/s* (columns (a) and (b)) for the <WKL$_1$, Crash> scenario, and *Pool of Nonpaged Bytes* (column (c)) for the <WKL$_2$, Hang> scenario (the complete result set can be found in (I. Irrera 2013)). Each row is a sample $\underline{C}(v_i)_{mj}$, and each column identifies the data fold in which the correlation value was obtained. The first row (in gray) is the pivot correlation distribution for the comparison. The correlation values equal to 1 are due to a numerical rounding of the representation (being in fact inferior to 1), or due to the very small number of failure events present in the fold. However, such inaccuracy allows a more clear analysis of the presented example. The values "-" mean that, for a given machine M_j., the number of folds created was less than 20 (as in the case of machine M_4 in Table 4.2 (b)). This is related to the limited number of failures observed.

Table 4.2 - An example of ROC-AUC correlation values relative to two variables

	(a) KW test passed		**(b) excluded from the test**		**(c) KW test passed**	
variable (scenario)	**Transition faults/s (WKL_1, CRASH)**		**Transition faults/s (WKL_1, CRASH)**		**Pool Nonpaged bytes (WKL_2 , HANG)**	
machine	**real (M_1)**	**XEN (M_2)**	**real (M_1)**	**VMWare Player (M_4)**	**real (M_1)**	**VirtualBox (M_5)**
ROC-AUC ($fold_i$)	1	0.994	1	0.98	0.86	0.835
	1	0.991	1	0.99	0.84	0.887
	0.92	0.969	0.92	-	0.85	0.85
	0.80	0.997	0.80	-	0.84	0.85
	1	0.98	1	-	0.84	0.86
	1	1	1	-	0.81	0.80
	1	0.981	1	-	0.87	0.87
	0.80	0.997	0.80	-	0.80	0.864
	1	0.988	1	-	0.88	0.85
	0.96	0.978	0.96	-	0.86	0.87
	1	0.966	1	-	0.89	0.86
	0.80	0.988	0.80	-	0.86	0.88
	0.96	0.988	0.96	-	0.86	0.84
	1	0.978	1	-	0.85	0.82
	0.88	0.97	0.88	-	0.86	0.87
	1	0.997	1	-	0.86	0.80
	1	0.98	1	-	0.88	0.85
	1	0.94	1	-	0.82	0.88
	0.80	0.981	0.80	-	0.87	0.90
	1	0.935	1	-	0.83	0.84

In the first case (Table 4.2 (a)), the symptoms correlation with failures of the variable *Transition Faults/s* coming from the real machine M_1 and the virtual machine M_2 (scenario <WKL_1, Crash>) is quite high and seem similar. In particular, the Kruskal-Wallis test could not reject the hypothesis of such symptoms to be similar, thus there is no evidence for considering the behavior of such variable not similar on both machines. On the other side, the set of correlation values relative to the same variable *and* in the same scenario collected from the machine M_4 (virtualized in a VMWare Player) is very small (only two values, Table 4.2 (b)). In this case, we decide to express no judgment on the similarity of the symptoms, also excluding the similarity hypothesis from being tested in such case.

Finally, the correlation values relative to the variable *Pool of Nonpaged Bytes* (Table 4.2 (c)), collected from the real machine and the virtual machine running on a VirtualBox environment, are also similar. Again, the Kruskal-Wallis test was not able to reject the hypothesis of the behavior of such variable being similar on both the systems.

4.6.3 Results and discussion

Table 4.3 presents the number of variables showing similar correlation between the original system and its virtualized copies, i.e., for which the Kruskal-Wallis test was not able to reject the null hypothesis. As we can see, the original machine M_1 shares with all its copies a subset of variables that have the same correlation with the failures observed (both for WKL_1 and WKL_2). Although the number of variables presented may seem small, it should be noted that the number of relevant variables for failure prediction is indeed quite small (G. Hoffmann and Malek 2006; Li, Vaidyanathan, and Trivedi 2002; Vaidyanathan and Trivedi 1999), which is a fact standing at the basis of feature selection. The results obtained for each couple of machines are quite similar, except for machines M_4 and M_5 in the case <WKL_1, Crash>. The very low number of crash failures observed in these two machines during the FIR limits the number of folds, and consequently the number of available correlation values, as exemplified in Table 4.2 (b). In this case, the Kruskal-Wallis test was not able to reject the null hypothesis, but the available correlation values are insufficient to have good confidence in the result. Thus we do not consider such results, assuming the null hypothesis being rejected. It is worth noting that we do not present results for the case <WKL_2, Crash>, as in this case we did not observe any Crash failures on the real machine.

Table 4.3 - Number of variables showing similar failure correlation

Machine	Workload	Failure Modes	
		Crash	*Hang*
M_1, M_2	WKL_1	47	16
	WKL_2	-	12
M_1, M_3	WKL_1	47	5
	WKL_2	-	6
M_1, M_4	WKL_1	0 (95)	27
	WKL_2	-	32
M_1, M_5	WKL_1	0 (96)	14
	WKL_2	-	12

An important aspect is that the set of variables that show similar correlation with failures *partially varies* when comparing the real system with the several virtualized versions. This means that the different characteristics of the *hypervisors* and the *hardware* have some impact on the original failure data. In fact, sets of variables from the virtualized environments only partially share variables, confirming the influence of the virtualization environments. Another important aspect is that the set of variables showing correlation with *Crash* failures is different from the set for the *Hang* case, and these two sets also differ when considering diverse workloads. The difference in the results for the two workloads is an evidence of the fact that the workload influences the failure prediction process (the OS is exercised in different ways making the fault activation pattern different), as also observed in the case study in Chapter 3.

4.6.4 Discussion on the impact of virtualization

There are two key aspects that should be emphasized based on the analysis in the previous section:

1) The fact that the sets of variables sharing the same failure correlation values on the original and each of the different virtualization technologies can be different, showing that **virtualization technology *does* influence the correlation of variables with failures**.

2) The existence of groups of variables sharing similar correlation values between the original system and its virtualized copies shows that **virtualization *can* be used to generate failure data**, and in particular that a subset of the failure data generated in a virtual environment is similar to failure date generated in the original target system and can be used in an equivalent manner. In practice, as variables with a high correlation with failures in the original system may not be as good in all virtualized systems, a previous analysis of the best virtualization technology for generating good datasets is needed.

Such results suggest that data may be generated from virtual copies of a system, but that **a preliminary study for identifying the variables in common is needed**. Moreover, given a set of variables to be used for failure prediction identified on the original system, a detailed analysis is needed for selecting the virtualization solution that shares such set of variables (or most of them) to be used to generate failure data.

4.7 Final remarks

In this chapter we addressed the challenge of using failure data generated in a virtualization environment hosting a copy of the system in which failure prediction is supposed to work. The proposed approach allows computing the similarity of failure data generated on a virtualized copy of a target system with respect to failure data generated on the original system, making use of the approach for generating failure data based on the injection of software faults proposed in Chapter 3. The approach can be used to validate if a given virtualization environment is adequate for generating failure data and also to guide the choice of a specific virtualization environment, when several alternatives are available.

The approach is based on the identification of the failure symptoms presented by the monitored variables, their correlation with the failures (and non-failure events, coming from Golden runs) observed in both the original and virtualized systems, and the comparison of the correlation values (of the individual variables) gathered from the different environments by using the Kruskal-Wallis statistical testing method. In practice, the approach includes three phases, starting from the generation of failure data using the solution proposed in Chapter 3, followed by the identification of failure symptoms and the correlation of such symptoms with the failures observed. The last phase is the assessment of the failure data similarity, based on the use of statistical tests.

In a case study, we demonstrated the applicability of the proposed approach and studied the similarity of failure data generated in a Windows-based system and four virtualized versions of it using diverse virtualization technology. Results shown that some sets of variables share similar symptoms across the original system and its virtualized copies and that the sets of variables in common between the original system and its virtualized versions are different. Hence, the virtualized environment influences the behavior of the collected variables when failures occur.

The next chapter presents a benchmarking approach for assessing and comparing alternative failure prediction models on a given target system. As choosing a failure prediction model is not trivial and requires large amounts of data, virtualization represents a key element for making the experiments possible and speeding-up the collection of failure data.

Chapter 5
Assessing and comparing Failure Prediction models

Effectively implementing failure prediction involves extremely accurate tuning, but also an adequate selection of the most suitable model (or models) for a particular system installation. Selecting a failure prediction model requires a rigorous assessment of alternative solutions using appropriate metrics, and their comparison using common datasets. However, this is a difficult task as the information about the performance of failure predictions models present in literature is not sufficient to choose a predictor for a particular target system. In fact, existing studies consider different systems, but the results are not comparable, and nothing can be said on how a given predictor will perform on a particular system installation. Also, many works in the literature provide incomplete and not comparable information, as so far there is no agreement on the best metrics to be used to assess the predictors, which vary from a study to another (Felix Salfner, Lenk, and Malek 2010).

Although several initiatives aiming at building failure data repositories have been taken (e.g., the Computer Failure Data Repository (Usenix and Carnegie Mellon University (CMU) 2006)), using such datasets is not sufficient for conducting a fair and sound comparison, as the assessment of failure prediction models with failure data collected from several systems does not allow taking into account the behavior of the system on which the predictors will run. This way, we argue that it is essential to collect failure data from the particular target system.

Advancing the state-of-the-art in failure prediction requires a systematic and rigorous approach for assessing and comparing alternative models, and such process must be supported by the generation of failure data. In this chapter we propose a **framework for benchmarking alternative failure prediction models**, making use of the failure data generation approach proposed in Chapter 3 and defining a procedure that assures a fair and sound assessment and comparison. In practice, we provide guidelines for implementing a procedure for benchmarking failure prediction models on a particular system (referred to as Failure Prediction Benchmark, or *FP Benchmark*), including choosing the adequate metrics for the assessment, the comparison of alternative models, and the validation of such results. Running the benchmark on the specific target system assures the results to be valid in the context of that system, as it takes into account its relevant characteristics (e.g., hardware, software, workload), thus minimizing the probability of harmful effects due to wrong estimated performance that may lead, for example, to wrong selection decisions. Obviously, the benchmark must ensure some key properties (M. Vieira and Madeira 2003; Gray 1993), namely the ease of use, ease of implementation, promptness, repeatability, portability, representativeness, and non-intrusiveness.

To demonstrate the proposed approach we also present a case study where the benchmark is used to assess and compare four models for predicting *Crash* and *Hang* failures in a machine running Windows XP (SP3). The case study demonstrates how the benchmark can be implemented and how the predictors can be assessed and compared in practice, based on their performance.

The chapter is organized as follows. Section 5.1 overviews the proposed benchmarking framework and discusses the properties that must be ensured. Sections 5.2, 5.3 and 5.4 describe the components of the benchmark, describe how to implement them, and outline the benchmarking procedure. Section 5.5 presents the case study, including the benchmarking results and a discussion on the benchmark properties. Finally, Section 5.6 summarizes the main lesson learned.

5.1 Overview of the approach and properties

Benchmarking is an experimental procedure that aims at providing a practical way to measure and compare properties of computer systems or components, ranging from performance (Gray 1993) to dependability and security aspects (Durães, Vieira, and Madeira 2004; M. Vieira and Madeira 2003). In practice, a benchmark reproduces the observations and measurements either deterministically or on statistical basis (giving confidence in the results obtained), and allows generalizing results to a limited extent (becoming useful beyond the particular case analyzed), attained by addressing the representativeness of the benchmarking process and components (Durães, Vieira, and Madeira 2004). According to (M. Vieira and Madeira 2003; Gray 1993), the concept of benchmarking can be summarized in three words:

1) *Representativeness*: a benchmark must include components (e.g., a dataset) that are representative of a given domain (in our case the failure prediction domain), thus reducing the distance between the benchmarked and the real environment (when present);

2) *Usefulness:* a benchmark must provide a useful representation of the entities under analysis, capturing the essential elements of the domain and characterizing their features, thus allowing one to use the results for choosing the best alternative or to guide improvement;
3) *Agreement*: a benchmark must specify a standard procedure to assess relevant measures related to an entity or a product on which users can agree, allowing measurement results to be accepted.

In this work we propose a framework for assessing and comparing failure prediction models, which we named *FP Benchmark*. The reasoning for proposing a *framework*, instead of directly refer to it as a benchmark, stands in the fact that a benchmark for prediction models (or machine learning algorithms in general) typically includes a workload (or dataset) (e.g., (Bache and Lichman 2013; Zheng 1993)). This workload is the data against which one or more systems are benchmarked. However, in the failure prediction domain, the dataset should include failure data that are specific to a particular system (i.e., the system were failure prediction should be done), and thus must be generated during the benchmarking process. In practice, we propose a benchmark that includes the generation of the dataset to assess and compare alternative predictors on a particular target system, but reference to it as a benchmark framework in order not to go against the terminology used in the field.

The proposed *FP Benchmark* includes three main components:

- **Dataset:** data needed to train and test the failure prediction algorithms. These data should mimic the behavior of the target system, taking into account the existing hardware and software components, the expected workload, the relevant failures, etc.
- **Metrics:** allow characterizing the effectiveness of the algorithms under benchmarking. The metrics must be easy to understand, allow the comparison among alternative algorithms from different points of view, and be generally accepted.
- **Benchmarking procedure:** rules that must be followed, including the set of phases that must be conducted, towards the calculation of the metrics.

For the generation of the dataset, the FP Benchmark adopts the approach proposed in Chapter 3, which includes a *workload* and a *faultload* for exercising the target system in a way that allows collecting failure data. Given the terminology used in existing performance (Zheng 1993; Gray 1993; M. Vieira and Madeira 2003) and dependability benchmarks (M. Vieira and Madeira 2003; J. Duraes, Vieira, and Madeira 2004), the term *dataset* is here intended as the workload to exercise the failure prediction models, while the term *workload* corresponds to the set of operations that the target system must execute for generating failure data. Thus, the *datasets* are composed of failure data coming from the target system, and it is the workload, in the classic machine-learning benchmarking nomenclature. The failure predictors are benchmarked using the datasets, and metrics are applied to their outputs (i.e., predictions) and behavior (e.g., time to train or predicting).

A benchmark for failure prediction models must address specific properties for the results to be sound, and to minimize inaccuracies due to the measurement procedure and the environment. For this reason, we adopted the recommendations from (Gray 1993; Durães, Vieira, and Madeira 2004; M. Vieira and Madeira 2003) about the properties a benchmark should envisage, namely:

1) **Ease of installation and use**: the benchmark should be composed of a simple program ready to be used or a document specifying how to implement the benchmark, where to find the tools needed (e.g., the fault injection tool, the monitoring tool, the workload), etc. In fact, a user must be able to analyze failure prediction models with the minimum effort possible.

2) **Promptness**: the benchmark execution should take the shortest time possible (preferably no more than a few hours per entity). Promptness increases the usability of the benchmark and of the failure prediction model, and potentially reduces the cost that one has to allocate for the failure predictor's benchmarking task.

3) **Non-intrusiveness**: the benchmark must require minimal or no changes in the entities under analysis, which in this context are the failure prediction models. Moreover, the target system cannot be influenced nor modified, as this may influence the generated datasets and thus invalidate the results and conclusions. If an alteration is not avoidable, this has to be controlled and reproduced for every failure prediction model under analysis, and taken into account when estimating the *representativeness* of the results.

4) **Portability**: the benchmark must allow comparing alternative failure prediction models in different domains and considering different types of target systems.

5) **Repeatability**: different executions of the benchmark in the same system must lead to the same results on a deterministic basis or in statistical terms. The results should not depend on a single execution of the benchmark: on the contrary, the benchmark must provide means for assessing a possible error in the performance results of the assessed tools.

6) **Representativeness**: the results coming from the benchmark must be representative of real world scenarios, i.e., the failure prediction models must behave similarly (in relative terms) when working on the target system in a real situation.

5.2 Dataset

To generate, collect and organize the datasets, the benchmark makes use of the failure data generation approach proposed in Chapter 3. In practice, it makes use of a labeled dataset, and a *k-fold* cross validation technique with a run-by-run (or run-wise) partition of Golden and Failure Data.

5.3 Metrics

Metrics are a fundamental part of the benchmarking process, as they serve for characterizing the characteristics of the predictor and for the user to fairly compare alternative models. According to (Gray 1993), benchmarking metrics must have the following properties: i) they should portray the relevant and key characteristics of the entity under benchmarking, ii) they must be easy to understand and use, and iii) they should be generally accepted. In addition, the measures must be easy to obtain without impacting the system behavior (the failure prediction models, in our case).

In this work we take a comprehensive approach and propose a large and extensible set of metrics, leaving to the benchmark user the selection of the relevant ones and the definition of new metrics (if needed), while guiding the choices done. In fact, in the same way we argue that the benchmark should be run in the system where failure prediction is being implemented (to take into account the system characteristics), we also defend that the outcome of the benchmarking process should fulfill the user needs. Thus, it is up to the user to select the metrics, although the benchmark framework provides guidelines and the support to calculate all the metrics presented below.

We propose metrics widely used to characterize the effectiveness of systems, particularly in the information retrieval and control systems area (see Table 5.1). For each family we discuss some guidelines for adopting the most adequate metrics for a fair comparison of failure prediction algorithms, based on practical experience and on the findings from (F. Salfner, Lenk, and Malek 2010). The proposed metrics are divided in four families, which allow analyzing the behavior of the failure prediction algorithms from different points-of-view:

1. Metrics based on the *contingency table (or confusion matrix)* (e.g., precision, recall, etc. (F. Salfner, Lenk, and Malek 2010));
2. Metrics based on *decision threshold* analysis (e.g., ROC, ROC-AUC, etc. (F. Salfner, Lenk, and Malek 2010));
3. Metrics on the *prediction error* (SSE, MSE, etc.);
4. Metrics related to *complexity* (training and testing time, etc.).

Table 5.1 - Recommended metrics for benchmarking failure prediction models.

	Metric	**Formula/Description**
Contingency table metrics	*Precision*	$\frac{TP}{TP+FP} = \frac{Correctly\ predicted\ failures}{All\ the\ predictions}$
	Recall / True Positive Rate / Sensitivity	$\frac{TP}{TP+FN} = \frac{Correctly\ predicted\ failures}{All\ the\ occurred\ failures}$
	False Positive Rate	$\frac{FP}{FP+TN} = \frac{False\ failures}{All\ the\ failure - free\ runs}$
	F-Measure	$\frac{2 \cdot Precision \cdot Recall}{Precision + Recall} = \mathrm{Mean}_H(Precision, Recall)$
Decision threshold analysis metrics	*ROC and ROC-AUC*	Plot of the True positive rate over False positive rate for various thresholds *(trade-off tpr/fpr respect to the decision threshold)*
Prediction error metrics	*Mean Square Error (MSE)*	$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2, Y \in \mathcal{R}^n$ $Y = vector\ of\ n\ true\ values$ $\hat{Y} = vector\ of\ n\ precitions$
Complexity metrics	*Set-up time*	The time needed to set up the model (e.g., training time for classifier models). The unit is "seconds per sample"
	Execution time	The time to perform the prediction over the sample taken at time *t*. The unit is "seconds per sample"

5.3.1 Prediction value-based metrics

These metrics are based on the *contingency table* (F. Salfner, Lenk, and Malek 2010) or *confusion matrix* (see Section 2.2.4), borrowed from the information retrieval field. In practice, we assume that any prediction falls into one of the following four cases:

1. *True Positive (TP)*: a failure is predicted, and a failure occurs in the expected time;
2. *True Negative (TN)*: no failure is predicted, and no failure occurs;
3. *False Positive (FP)*: a failure is predicted, but there is no actual failure in the expected time;
4. *False Negative (FN)*: the predictor does not predict any failure, but a failure actually occurs.

More complex metrics can be defined by combining the four cases above in a different manner (Van Rijsbergen 1979), including:

- **Precision**: the ratio of correctly predicted failures with respect to the number of all predicted failures. In our context it can be represented as follows:

(5.1) $$Precision = \frac{TP}{TP + FP} = \frac{Correctly\ predicted\ failures}{All\ the\ predictions}$$

- **Recall**: a ratio of correctly predicted failures with respect to the number of true failures. In our context it can be represented as follows:

(5.2) $$Recall = \frac{TP}{TP + FN} = \frac{Correctly\ predicted\ failures}{All\ the\ occurred\ failures}$$

- **F-Measure**: the weighted harmonic mean (used to average rates) between Precision and Recall, assuming equal weights:

(5.3) $$F - Measure = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} = \mathrm{Mean}_H(Precision, Recall)$$

It is important to emphasize that the benchmark user should take into account the following aspects when adopting metrics based on the confusion matrix:

1. Predicting a failure is more important than predicting the absence of a failure, thus *TP must be maximized;*

2. False negatives must be avoided, as failures can have dramatic consequences, thus *FN must be minimized*;
3. *FP must be minimized*, as being continuously signaling failures that do not occur is not really predicting.

5.3.2 Decision threshold-based metrics

These metrics are based on the analysis of the predictor behavior when changing a threshold (i.e., the decision threshold). As in most prediction systems, the *failure/no-failure* prediction is done by applying a threshold to the (numeric) output of the predictor: if the output is above (or below) the threshold, a failure (or no failure) is predicted. Usually, this threshold is adjustable and different values lead to different behaviors of the prediction model, impacting also the values of TP, FP, FN, and TN, from which several other metrics can be defined. For example, plotting the values of the *True Positive Rate* (TP/(TP + FN), or Sensitivity) against the *False Positive Rate* (FP/(TN + FP), or 1-Specificity) while varying the threshold allows obtaining the *Receiver Operating Characteristic* curve, or ROC (Fawcett 2006), and its area (ROC-AUC), which is a widely used metric (see Section 2.2.3). Other examples are the F-Measure (Hand 2012) and Precision-Recall curves (Felix Salfner, Lenk, and Malek 2010).

Threshold analysis is a widely accepted method for assessing the performance of binary classifiers (Fawcett 2006). It is also considered as an effective way to overcome the problem of evaluating classifiers when using imbalanced datasets (e.g., (Chawla, Japkowicz, and Kotcz 2004; Chawla 2010)), as is the case in the failure prediction scenario, where no-failure labels outnumber the labels relative to failures (see data labeling in Section 3.4). In fact, when a dataset contains more positive than negative samples (or vice-versa), a classifier with a fixed-threshold may present a poor performance (Chawla, Japkowicz, and Kotcz 2004), which can be improved by changing the threshold. Several works demonstrate that threshold-based performance analysis is *independent of the class priors* (i.e., the distribution of the samples belonging to each class) (e.g., (Chawla, Japkowicz, and Kotcz 2004; Chawla 2010; Zweig and Campbell 1993; Fawcett 2006; Wang 2008)).

Among all the decision threshold-based metrics presented, for the failure prediction context we propose the use of ROC analysis due to some significant characteristics, namely: independence from the dataset, capability for performing sensitivity analysis in the context of varying thresholds, easiness of interpretation of the results, and large usage for the assessment of information retrieval systems.

5.3.3 Prediction error-based metrics

The metrics in the third family focus on the prediction error and are widely used in the systems control field to compare the effectiveness of different models. In the context of benchmarking failure predictors, they can be used to characterize models whose output is a class or element (e.g., binary classifiers) or models with a numerical real output (e.g., regression models) (F. Salfner, Lenk, and Malek 2010) before their output is compared to a threshold for producing such class (e.g., a 1 is obtained by checking if the output is above a given threshold, else it is 0). In this case, these metrics can complement TP, FP, etc., as they include information on the prediction error of each predictor.

One of the metrics that can be considered is the Mean Square Error (MSE), as it is useful to assess the quality of a predictor as an estimator of the occurrence of a failure (in terms of probability, time-to-failure, etc.). The definition of the MSE in this context is as follows:

$$MSE(Predictor) = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2, \quad (5.4)$$

$$Y = vector\ of\ n\ true\ failures$$

$$\hat{Y} = vector\ of\ n\ predicted\ failures$$

5.3.4 Time complexity metrics

The metrics belonging to the fourth family are related to the time the models take to be trained and optimized (Set-up time) and to perform the prediction (Execution time). This is of utmost importance in some scenarios, as there is often a trade-off between the performance of an algorithm and the time and resources needed to run it. Obviously, the time measured strongly depends on the machine that hosts the algorithm (which may not be the target system).

5.4 Procedure

Benchmarking requires a rigorous procedure, driving the user from the predictors' assessment to the comparison of the results. Obviously, the failure prediction models under benchmarking must share the same training and testing sets, as only in this way the final results can be comparable. The proposed benchmark procedure is depicted in Figure 5.1, and includes six phases:

1) **Preparation**: this phase consists of identifying the set of *parameters* for benchmarking the failure prediction models (e.g., the metrics to consider, the failure modes to predict, the intervals relative to the failure prediction task), and selecting and installing the *failure prediction models* under benchmarking.

2) **Data generation and dataset building**: the dataset is built from the data collected, organizing the data and associating the information about failure/no-failure, according to the approach proposed in Chapter 3. Datasets are divided in *training* data (to train the algorithms), *testing* data (to test their prediction ability), and validation data (for evaluating the prediction generalization). Each dataset is relative to a specific *scenario*, identified by the benchmark parameters such as the workload, the failure mode, or the prediction time, which influence the system dynamics (reflected in the datasets) and may call for a different prediction model.

3) **Execution of the prediction algorithms**: each algorithm must be trained using a training dataset, while the testing and validation datasets should be used for evaluation. In this phase, the output of each predictor is collected for later processing (to calculate the relevant benchmark metrics). This phase can be divided in three parts: i) **training**, where the failure prediction algorithm is trained using labeled data (*training dataset*) for discriminating *failing* from *non-failing* situations; ii) **prediction**, where each failure predictor tries to label a set of unlabeled data (*testing dataset*); and iii) **output collection**, where the outputs of each failure predictor (i.e., the labels suggested) are collected. In order to obtain a sound assessment and comparison, additional tasks may be needed such as feature selection, failure prediction optimization, and validation. The following tasks are vertical to the training and prediction tasks, and are usually performed when training a prediction model:

    a. **Features selection**: the dataset includes several variables (features). However, features do not have the same importance (H. Liu and Yu

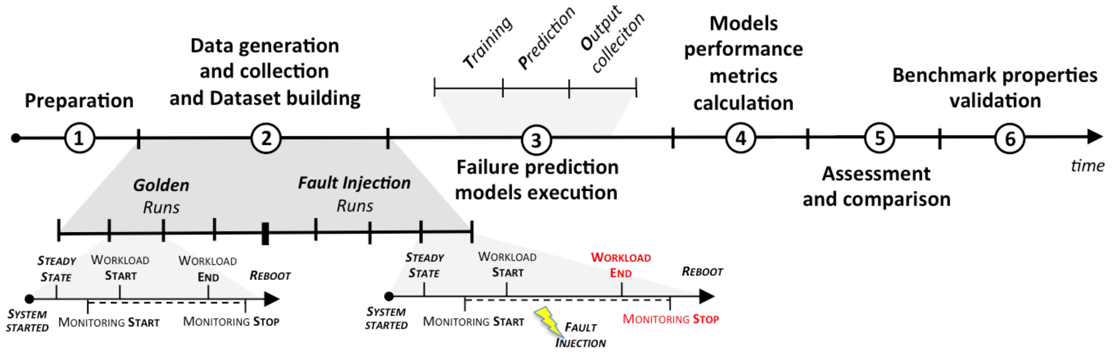


Figure 5.1 - The benchmarking procedure using fault injection at runtime

2005), and reducing their number frequently increases the prediction quality. Well-known methods for feature selection are *forward selection* (starting from a subset, add features until the predictor performance starts decreasing), and *backward elimination* (starting from the entire set, exclude features).

b. **Parameters optimization**: predictors are trained by fixing several parameters, which impact on their performance. Each predictor should be assessed using the set of values that maximizes its performance. Several techniques for finding the optimal working parameters (e.g., number of neurons for a neural network) can be found in literature (e.g., (Bishop and others 2006; Kennedy 2010; Schölkopf and Smola 2002)).

4) **Performance metrics calculation**: the predictions performed by each model under benchmarking are processed to obtain a value for the chosen metrics. For instance, to compute the Precision of a predictor, the labels produced (i.e., the *predictions*) and the expected labels (*testing data* labels) are compared, a confusion matrix is created (TP, TN, FP, and FN are computed), and the Precision is finally obtained. Each predictor should also be evaluated using the validation datasets, as one must assure that the performance achieved does not depend on a particular dataset (i.e., results are not biased). This property (i.e., to which extent the results will hold in the operational scenario, or when using a different dataset) is closely related to the confidence one may have in the benchmarking results. Validation techniques should be used to assess the confidence on the results: an example is *k-fold cross validation*, as showed in the previous chapters.

5) **Assessment and comparison**: the user analyzes the benchmark results and selects the failure prediction algorithm that best fits its requirements. Although not mandatory, the user can use techniques for non-subjective analysis of benchmark results, as the one proposed in (Martinez et al. 2014).

6) **Benchmark properties validation**: this last step consists of assessing the fundamental properties of the benchmark, thus validating the assessment and comparison results. The properties to validate are the ones presented in Section 5.1. In particular:

   a. *Ease of installation and use*, *promptness*, *non-intrusiveness* and *portability* can be validated through the analysis of the installation and usage process during the benchmarking campaign;

   b. *Repeatability* can be automatically validated through the use of a validation technique, as the *k-cross fold validation*;

c. *Representativeness* can be validated by analyzing the accuracy of generated failure data and of the benchmark results, by making use of the *indirect* (or weak) *accuracy estimation* proposed in Section 3.5.2. In fact, accurate benchmark results are a sufficient condition for the benchmark to be representative.

5.5 Case Study: Benchmarking different failure prediction models

In this section we present a case study to demonstrate the process of building a *FP Benchmark,* to analyze its applicability in benchmarking alternative predictors to be used on a particular system, and to validate its properties. It is important to emphasize that the case study serves only to demonstrate the approach proposed. This means that, although realistic, the choices about the workload, the failure modes, the failure prediction algorithms, etc., might not be not the most adequate ones for real world scenarios.

The case study aims at assessing and comparing the performance of failure prediction systems based on a SVM classifier (Support Vector Machine, (Cortes and Vapnik 1995)), predicting failures in a computer system running Windows XP SP3 OS (i.e., the target system). Four different flavors of a Support Vector Machine are benchmarked, including predictors implementing the sliding window technique proposed in Section 3.6:

- **FPA$_1$** - SVM classifier (Gaussian kernel);
- **FPA$_2$** - SVM classifier (Gaussian kernel) + sliding window;
- **FPA$_3$** - SVM classifier (Linear kernel);
- **FPA$_4$** - SVM classifier (Linear kernel) + sliding window.

A representation of the *FP Benchmark* is shown in Figure 5.2. The target system is a virtualized Windows OS running on top of a VMWare vSphere server (Frappier 2014), executing two workloads (WinRAR (RAR Lab) and COSBI OpenSourceMark ("COSBI OpenSourceMark")). The benchmark is implemented on an analysis/controller system, in charge of controlling the experiments and analyzing the failure data coming from the system, while fault injection and data collection is performed on the target system. The fact that the target system is a virtualized machine allows simplifying the management of the experiments, permitting easily starting/stopping/rebooting the target system, as well as restoring its internal state after each fault injection run, following the recommendations in Section 3.3. It is worth noting that in this case study we do not focus on examining the accuracy of

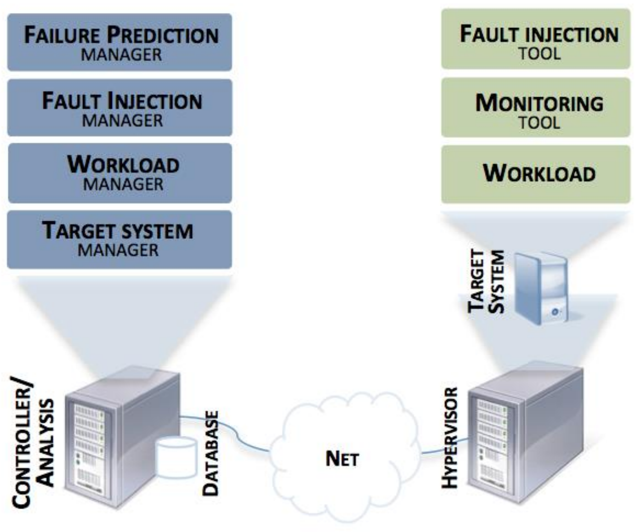


Figure 5.2 – FP Benchmark components deployed.

the failure data generated using a virtualized environment, as the original target system is also a virtualized system, hence the failure prediction models will work directly on information coming from a virtualized environment.

The failure prediction models were run on the analysis machine, to isolate them from the target system environment. In practice, the configurations of the machines are the following:

- **Machine #1 (target system)**: Intel i5-650@3.60GHz machine, 8GB RAM, running a Windows XP OS (SP3) in a VMWare vSphere server based on ESXi v5.0.
- **Machine #2 (analysis/controller system)**: Intel i5-650@3.60GHz machine, 8GB RAM, running a Windows XP OS (SP3).

For implementing the benchmarking procedure we used the PowerShell scripting language for Windows OS environments (Siddaway 2012). This scripting language permitted, in particular, a fast prototyping and easy implementation of functionalities, with the native and easy use of network communication. The analysis of the benchmarking results is conducted in the analysis machine by using the MATLAB environment, which supports the manipulation of data related to the failure prediction, such as the datasets, the prediction results, and so on.

The failure prediction models (based on SVM) are based on the ones implemented by the libSVM C/C++ libraries (Chang and Lin 2011), which provides interfaces for Windows- and Linux-based operating systems, as well as utility programs for the analysis of the prediction results for MATLAB and similar environments.

About the data generation and collection, we selected the failure modes (*Crash* and *Hang*) and tools used in the case study in Chapter 3. Recalling, we monitored 233 numerical variables representing the state of the OS resources, at the sample rate of one value per second, using the Logman tool that is included in Windows OSs family. A three-step feature selection was used to select the set of variables that maximizes the performance of the prediction system. The resulting set consists of 25 variables out of the initial 233 (see Table 5.2), for each scenario <WKL, Failure>, and for each couple of values (Δt_l, Δt_p) considered in the analysis. Again, we adopted the G-SWFIT software fault injection tool and injected faults in the code of the dynamic libraries (the *kernel32.dll* and *ntdll.dll* system library modules) used by the system process *svchost.exe* (a more detailed description is in Section 3.3). The injection in *kernel32.dll* and *ntdll.dll* modules leads to different datasets.

The number of experiments (i.e., fault injection runs) needed to benchmark the failure prediction models was calculated in order to have enough failures to train, test and validate the models. In practice, we analyzed the activation rate (i.e., a fault eventually causing a failure) in the experiments presented in the previous chapters, during which we observed an average activation rate of 2%. Thus, for causing one

Table 5.2 - Selected variables, for (WKL_1, Crash) and (Δt_l,Δt_p)=(10s, 5s)

Variable ID	Variable name	Monitored component
106	% Committed Bytes In Use	Memory
115	Demand Zero Faults/sec	Memory
123	Pool Nonpaged Allocs	Memory
109	Available Mbytes	Memory
117	Page Faults/sec	Memory
125	Pool Paged Allocs	Memory
128	System Cache Resident Bytes	Memory
127	Pool Paged Resident Bytes	Memory
201	C2 Transitions/sec	Processor
192	% C2 Time	Processor
209	Exception Dispatches/sec	System
220	System Calls/sec	System
210	File Control Bytes/sec	System
154	Avg. Disk sec/Write	PhysicalDisk
156	Current Disk Queue Length	PhysicalDisk
152	Avg. Disk sec/Read	PhysicalDisk
155	Avg. Disk Write Queue Length	PhysicalDisk
94	Avg. Disk sec/Transfer	LogicalDisk
139	Semaphores	Objects
182	Pool Nonpaged Bytes	Process
173	IO Other Bytes/sec	Process
39	Sync Data Maps/sec	Cache
26	Data Map Pins/sec	Cache
16	Async Copy Reads/sec	Cache
224	% User Time	Thread

hundred failures, we decided to execute each workload at least 3000 times. The datasets are organized as proposed in Chapter 3: each dataset is divided in *training* (to train the model) and *testing* data (to test its prediction ability), using a 50% ratio. The data are labeled based on different *(Δt_l, Δt_p)* parameters.

The metrics chosen for assessing the failure prediction algorithms are the *ROC-AUC*, the *F-Measure* (F. Salfner, Lenk, and Malek 2010) (considering the optimal threshold point found with the ROC method), the Set-up time (i.e., the time needed for a predictor to be trained), and the AUC/Set-up time ratio.

5.5.1 Benchmarking campaign

Table 5.3 summarizes the benchmarking parameters, including the failure prediction models, the failure modes, the variable selection method, and so on. A tuple of parameters <Dataset length, Workload, Failure *Mode, Δt_l, Δt_p*> is associated to each benchmark run, resulting in a total of 100 runs, in which the predictors are benchmarked. The combination <Workload, Failure mode> allows defining four different scenarios for the analysis: <WKL_1, Crash>, <WKL_1, Hang>, <WKL_2, Crash>, and <WKL_2, Hang>. The failure prediction parameters we considered include predictions lead-time *Δt_l* from 10s up to 50s in advance, with a maximum prediction interval *Δt_p* of 25s. The windowing value for the FPA_2 and FPA_4 is between 2s and 10s.

Details about the failures generated are presented in Table 5.4. We should highlight the fact that the failure occurrence is, as expected, 2% in average, the same activation rate obtained in our case studies presented before. The entire benchmarking campaign took about one month. In the next sections we present and discuss the benchmarking results from multiple perspectives.

Table 5.3 - The details of the analysis

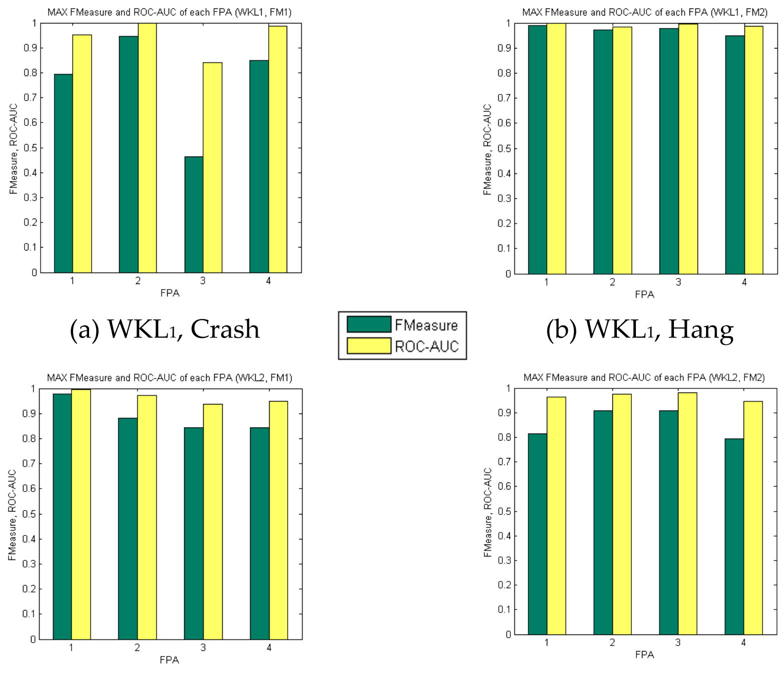
Parameter	Values
Failure Modes	Crashes, Hangs
Workloads	WKL_1 (WinRAR), WKL_2 (COSBI OpenSourceMark)
Predictor	SVM (Gaussian kernel)
Variable selection	Backward elimination + wrapper approach
Predictor Optimization	Grid search (gross) + Deepest descend (fine)
(γ, C) (Grid search)	γ= [2^{-10},1], C=[2^{-1}, 2^7]
Δt_l (Failure prediction)	10, 20, 30, 40, 50 s
Δt_p (Failure prediction)	5, 10, 15, 20, 25 s
Window size (w)	2,3,4,7,10 s
Results validation	5-folds cross validation

Table 5.4 - Failures generated

Workload	**# Golden Runs**	**# Fault Injection Runs**	**Failures detected**		
			Total %	***System Crash %***	***System Hang %***
WKL$_1$	500	3000	121 *(4.03%)*	46 *(1.53%)*	75 *(2.5%)*
WKL$_2$	500	3000	74 *(2.47%)*	6 *(0.2%)*	68 *(2.27%)*

5.5.2 Best performing Failure Prediction model

The first analysis is focused on the **absolute performance of the predictors**, i.e., not considering the performance relative to a specific (Δt_l, Δt_p). The results are shown in Figure 5.3. The first bar of each couple represents the F-Measure of a single predictor, and the second its ROC-AUC. Each of the four plots is relative to one of the four operational scenarios. In the first scenario <WKL$_1$, Crash>, the predictor with the highest F-Measure (and ROC-AUC) is FPA$_2$, i.e., the SVM classifier with a sliding window and a Gaussian kernel. In the second scenario <WKL$_1$, Hang>, the best performance was obtained by FPA$_1$ (SVM classifier with a Gaussian kernel) both in terms of ROC-AUC and F-Measure. This same algorithm has the best performance in the third scenario <WKL$_2$, Crash> scenario. Finally FPA$_3$ (SVM classifier with a linear kernel) performs better than the others in the last scenario <WKL$_2$, Hang>.



(a) WKL$_1$, Crash

(b) WKL$_1$, Hang

(c) WKL$_2$, Crash

(d) WKL$_2$, Hang

Figure 5.3 - FPA with the highest F-Measure/ROC-AUC.

It is worth noting that the performance (F-Measure and ROC-AUC) is an average of the performance of each predictor on each of the partial datasets coming from the *k-fold* cross validation, and that similar conclusions can be obtained considering either the F-Measure or the ROC-AUC. Moreover, each performance value is the maximum observed among the parameters (Δt_l, Δt_p), i.e., the performance is the best achievable. This makes sense, as a failure prediction algorithm should be used for a specific couple of values (Δt_l, Δt_p). In this case, the predictor performance is maximized by the parameters (Δt_l^*, Δt_p^*). In the same way, FPA_2 and FPA_4 performance is the maximum value for the couple (Δt_l, Δt_p) and the windows width *w*.

The fact that there are different best predictors for different scenarios suggests an influence of the workload in failure prediction. Recalling the observations about the use of the sliding window in Chapter 3, we can confirm that such technique still outperforms classifiers that do not include the time dimension in several cases.

5.5.3 Best Failure Prediction model for each couple (Δt_l, Δt_p)

The objective of the second assessment is to **select the best predictor given a prediction horizon** (Δt_l, Δt_p). This scenario may emerge when one needs a failure predictor with specific characteristics in terms of prediction lead-time, for instance due to the target system characteristics. Table 5.5 shows the best failure prediction algorithm for each couple (Δt_l, Δt_p) and the relative average performance in terms of the F-Measure. The results relative to ROC-AUC are omitted as they lead to similar conclusions. Again, the best performing failure prediction algorithm varies for each scenario and for each prediction horizon (Δt_l, Δt_p). The most interesting scenario is <WKL_2, Hang>, where a different predictor (among FPA_1, FPA_2 and FPA_3) is proposed for each couple (Δt_l, Δt_p), which is coherent with the results in the previous section.

The results in Table 5.5 may help in the design phase of a failure prediction system, to better select the prediction horizon (Δt_l, Δt_p). Moreover, at runtime a meta-predictor can be used for selecting one among the results provided by the predictor for each (Δt_l, Δt_p) horizon, thus achieving the highest prediction performance. It is also worth noting that failure prediction can generally follow two approaches. In a first one, the values of Δt_l and Δt_p are defined by the needs of the system (e.g., a web server would need a prediction at least 1 minutes in advance, to have time to save its internal state) or by contract (e.g., on a Service Level Agreement). The second approach consists of having several values, resulting in a "best-effort" approach (e.g., train several predictors considering the multiple combinations of Δt_l and Δt_p and choose the one that performs better) (Ivano Irrera, Pereira, and Vieira 2013).

Table 5.5 - Predictors F-Measure relative to prediction parameters Δt_l and Δt_p

Δt_p \ Δt_l	10s	20s	30s	40s	50s
5s	FPA2, 0.549	FPA2, 0.611	FPA2, 0.696	FPA2, 0.747	FPA2, 0.854
10s	FPA2, 0.696	FPA2, 0.747	FPA2, 0.854	FPA2, 0.879	FPA2, 0.894
15s	FPA2, 0.854	FPA2, 0.879	FPA2, 0.894	FPA2, 0.917	FPA2, 0.936
20s	FPA2, 0.894	FPA2, 0.917	FPA2, 0.936	FPA2, 0.944	FPA2, 0.906
25s	FPA2, 0.936	FPA2, 0.944	FPA2, 0.906	FPA2, 0.897	FPA2, 0.894

(a) WKL$_1$, Crash

Δt_p \ Δt_l	10s	20s	30s	40s	50s
5s	FPA1, 0.877	FPA1, 0.921	FPA1, 0.930	FPA1, 0.936	FPA1, 0.934
10s	FPA1, 0.930	FPA1, 0.936	FPA1, 0.934	FPA2, 0.947	FPA1, 0.969
15s	FPA1, 0.934	FPA2, 0.947	FPA1, 0.969	FPA2, 0.963	FPA2, 0.965
20s	FPA1, 0.969	FPA2, 0.963	FPA2, 0.965	FPA2, 0.963	FPA2, 0.970
25s	FPA2, 0.965	FPA2, 0.963	FPA2, 0.970	FPA1, 0.973	FPA1, 0.990

(b) WKL$_1$, Hang

Δt_p \ Δt_l	10s	20s	30s	40s	50s
5s	FPA1, 0.877	FPA1, 0.921	FPA1, 0.930	FPA1, 0.936	FPA1, 0.934
10s	FPA1, 0.930	FPA1, 0.936	FPA1, 0.934	FPA2, 0.947	FPA1, 0.969
15s	FPA1, 0.934	FPA2, 0.947	FPA1, 0.969	FPA2, 0.963	FPA2, 0.965
20s	FPA1, 0.969	FPA2, 0.963	FPA2, 0.965	FPA2, 0.963	FPA2, 0.970
25s	FPA2, 0.965	FPA2, 0.963	FPA2, 0.970	FPA1, 0.973	FPA1, 0.990

(c) WKL$_2$, Crash

Δt_p \ Δt_l	10s	20s	30s	40s	50s
5s	FPA3, 0.907	FPA2, 0.896	FPA2, 0.901	FPA2, 0.907	FPA2, 0.897
10s	FPA2, 0.901	FPA2, 0.907	FPA2, 0.897	FPA2, 0.875	FPA2, 0.870
15s	FPA2, 0.897	FPA2, 0.875	FPA2, 0.870	FPA2, 0.855	FPA3, 0.853
20s	FPA2, 0.871	FPA2, 0.855	FPA3, 0.853	FPA3, 0.851	FPA2, 0.845
25s	FPA3, 0.853	FPA3, 0.851	FPA2, 0.845	FPA2, 0.836	FPA3, 0.836

(d) WKL$_2$, Hang

5.5.4 Performance *vs* Computational cost

The benchmark also allows other interesting analysis, as on the **computational cost of the best performing failure prediction algorithm**. In Figure 5.4 we compare the failure prediction algorithms on the basis of the ROC-AUC, the Set-up time, and the ratio AUC/Set-up time. The Set-up time actually represents the computational cost of the algorithm and is shown in the plot as a relative value, i.e., divided by the maximum average Set-up time obtained during the benchmarking campaign (this is just a rescaling operation that does not influence the observations).

In scenarios <WKL$_1$, Crash> and <WKL$_1$, Hang>, the failure predictor that showed the best AUC/Set-up time ratio is FPA$_1$ followed by FPA$_2$. This was expectable, as FPA$_1$ and FPA$_2$ are the best performing predictors in these scenarios, and training an SVM with a linear kernel takes much less time than training a SVM with a Gaussian or other kinds of non-linear kernels. In scenario <WKL$_2$, Crash> the best ratio was obtained by FPA$_3$, and FPA$_1$ performed best in the case of *hang* failures. The results in terms of AUC/Set-up time ratio make evident the fact that, in general, the gain obtained in terms of performance is not justified by the computational cost that comes with it. If the cost in terms of Set-up time has to be taken into account, the performance/computational cost ratio should be considered as an input of the selection decision.

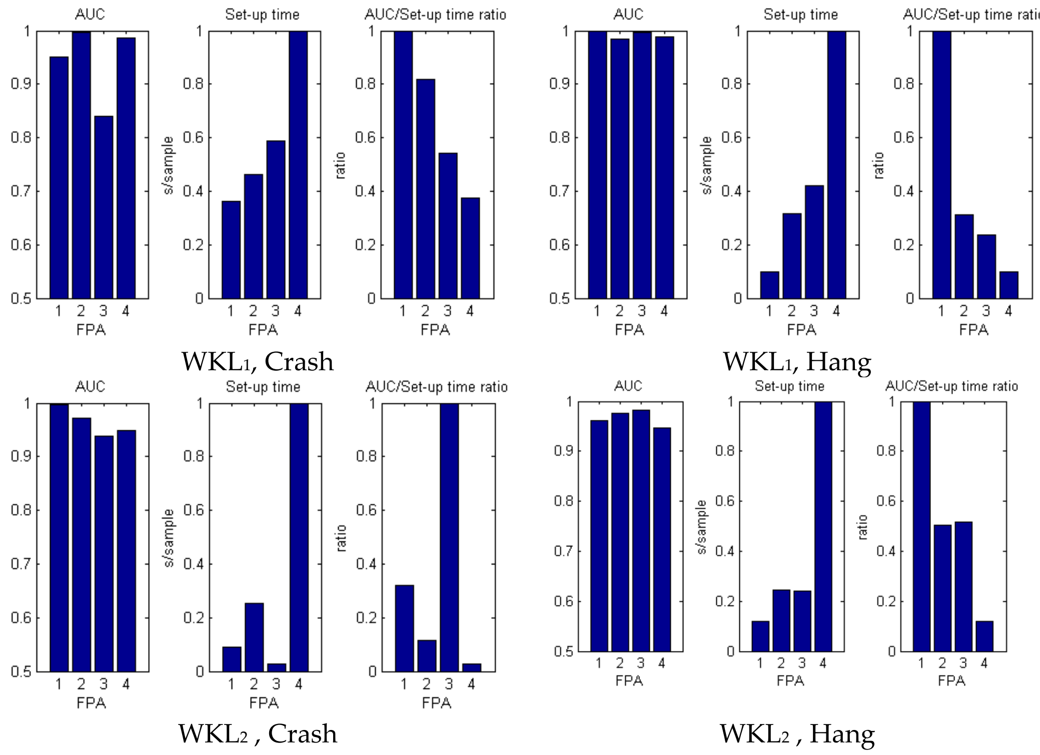


Figure 5.4 - Performance *vs* Computational cost

5.5.5 Properties of the implemented benchmark

In this section we discuss the concrete properties of the *FP Benchmark*, analyzing its implementation and usage. In some cases it was necessary to experimentally validate the benchmark properties (e.g., repeatability) and in the other cases the validation is based on reasoning (e.g., easy to implement and use).

A benchmark that is not **easy to implement and use** is clearly unacceptable. Our benchmark is implemented using PowerShell scripting, C++ libraries and the MATLAB environment, largely adopted and easy to use solutions. Our benchmark is completely automatic, fast and based on simple procedures. In particular, although complex, the fault injection is facilitated by existing tools of easy use, as the G-SWIFIT tool (J. A. Duraes and Madeira 2006).

About the **promptness** in obtaining the benchmarking results, using fault injection we were able to cause 195 failures in less than 800 hours. We tried to compare such statistics against Windows XP OS failures data available online, which are rather rare and incomplete. Nevertheless, a study on Microsoft products mentions that Windows XP OS has a 600 hours MTTF, although this value considers both OS and application failures (“Tech Insider - Various Studies”). In this perspective, even considering both OS and application failures, using fault injection reduces the collection of failure data by two orders of magnitude.

For addressing the property of the benchmark to be **not intrusive**, the predictors' control, execution, results collection and analysis are processed by a separate system (*controller* or *analysis system*), which assures *non-intrusiveness* on both the prediction models behavior and the target system behavior. Furthermore, the benchmark does not require any kind of modification on the failure prediction models, as they simply use the dataset generated from the target system for training and testing purposes. On the other hand, the fault injection tool is intrusive, but this is something we cannot avoid considering the proposed technique for generating failure data.

A benchmark must allow comparing different tools in different domains and for different types of systems, thus being **portable**. In our case, this is a property that applies to the benchmarking framework and not to a concrete implementation (the implemented benchmark can only be used in the target system for which the tools were developed). In practice, the more general the benchmarking components are (i.e., their capability of being implemented on every kind of architecture, or whose implementation is hardware-independent), the more the benchmarking process is portable. In our case, every component of the benchmark is portable, and the benchmark can be used to assess and compare any kind of failure prediction model. In particular, the workload can be implemented on any software system and the faultload and fault injection tool can be defined for different types of systems following the recommendations in (J. A. Duraes and Madeira 2006).

Table 5.6 - ROC-AUC distribution along the dataset folds (excerpt).

Failure prediction algorithm	*(Δt_l, Δt_p)*	ROC-AUC values (folds)				
FPA_1	(30s, 5s)	0,96	0,93	0,96	0,86	0,96
FPA_2	(10s, 5s)	0,96	0,78	0,94	0,79	0,95

When run more than once over the same failure prediction model, the benchmark must report the same results (at least in statistical terms). **Repeatability** is a fundamental property, as each execution of the benchmark should give confidence about the results obtained. A necessary condition that the benchmark addresses is keeping the faultload and the workload parameters constant, as well as restoring the target system state at the beginning of each fault injection run. The results from applying the *k-fold cross validation* to the dataset also gives to us some insights on how the results vary. Table 5.6 presents the ROC-AUC measurements relative to two of the failure prediction algorithms benchmarked considering different dataset folds. The results are relative to the best ROC-AUC values in the case <WKL_1, Crash> and we can observe that the distribution of the values does not vary considerably in the case of the FPA_1, while the FPA_2 there are some variations, which are most likely due to the prediction values chosen (Table 5.5 (a) confirms that FPA_2 had poor performance using (Δt_l, Δt_P)=(10s, 5s))). Such results suggest that the benchmark is repeatable.

Finally, the results the benchmark should be **representative** of real world scenarios, as only in this case they may be considered relevant. In this particular scenario, representativeness also depends on the accuracy of the failure data. Necessary conditions for generating accurate data are the injection of realistic software faults, i.e., software faults that are likely to be found in real systems (J. A. Duraes and Madeira 2006), and the representativeness of the workload used for generating failure data, as presented in Chapter 3. On the other hand, a sufficient condition for the benchmark results to be representative is their accuracy with respect to results obtained in a real scenario. We here validate the benchmark representativeness by using the *weak accuracy estimation* analysis, as it involves the analysis of performance of the predictors. Thus, we used the approach already introduced in Chapter 3, building two datasets based on the injection of faults in two different modules: *kernel32.dll* (Dataset#1) and *ntdll.dll* (Dataset#2). Here we use the relative performance estimation metric ε_μ\* applied to the mean values of the predictors, which is calculated as the relative error between the performance obtained using two different datasets as defined by equation (3.4).

Table 5.7 shows the results relative to the best performing failure prediction algorithm for each scenario and each (*Δt_l, Δt_p*), together with the relative performance error ε_μ\*. The synthetization error estimates ε_μ\* show a value between about 0,2% and 5%, in terms of ROC-AUC, which suggests an error of at most 5% in terms of ROC-AUC in predicting in a real (or at least different) operational scenario.

Table 5.7 - ROC-AUC and synthetization error.

	WKL_1				**Crash**	WKL_2			
	FPA_1	***FPA_2***	***FPA_3***	***FPA_4***		***FPA_1***	***FPA_2***	***FPA_3***	***FPA_4***
Dataset#1	0,963	0,976	0,960	0,825		0,998	0,986	0,996	0,978
Dataset#2	0,975	0,995	0,938	0,784		0,997	0,980	0,995	0,979
ε_μ\ (%)*	***1,2***	***1,9***	***2,3***	***5,0***		***0,1***	***0,6***	***0,1***	***0,05***
	WKL_1				**Hang**	WKL_2			
Dataset#1	0,973	0,946	0,850	0,876		0,994	0,970	0,929	0,907
Dataset#2	0,885	0,927	0,852	0,834		0,968	0,923	0,925	0,869
ε_μ\ (%)*	***8,8***	***2,0***	***0,2***	***4,8***		***2,6***	***4,8***	***0,4***	***4,2***

Such results imply that the benchmarking representativeness can be improved, aiming at reducing the error estimate ε_μ^* to zero by improving the several components and parameters used. Benchmarking results with a given ε_μ^* can still be used, however the user must consider that the measured performance value of a given failure predictor FPA_i has an associated risk value corresponding to ε^*_{FPAi}. The results also show that the algorithms ranking keeps in the validation, except in scenario <WKL_1, Hang>, where FPA_2 performs better than the expected best-performing FPA_1. However, this has been assessed as a side effect due to the reduced number of failures available to validate the results.

5.6 Final remarks

In this chapter we proposed a benchmarking framework for a sound assessment and comparison of alternative failure prediction models on a particular target system. The framework is based on the failure data generation approach proposed in Chapter 3, making use of realistic software fault injection, and allows an easy and prompt analysis of the prediction models. We provided guidelines for the implementation of the benchmark and discussed the relevant properties, such as being simple to implement and to use, being fast in its execution, being portable and non-intrusive, and providing results that are repeatable and representative.

The proposed benchmarking framework includes three key components, namely the dataset (build using the approach in Chapter 3), the benchmarking metrics and the analysis procedure. In particular, we proposed four families of metrics for taking into account the properties of a failure prediction model to be fairly assessed, which enables the benchmarking results to be generally accepted.

The proposed benchmark was used in a concrete case study based on the Windows XP OS. Two different workloads and two failure modes were considered, and four failure predictors were assessed and compared under different scenarios. The results showed that the procedure could be used for training and testing failure predictors in a cost-effective and easy way. The properties that a benchmark must ensure were discussed and validated. In particular, we assessed the representativeness of the results by estimating the *accuracy* of the generated failure data by using the weak synthetization metrics.

The next chapter presents a framework for addressing the problem of using failure prediction models in evolving systems, in which there may be a need for re-training a failure predictor after a change affects the target system or its environment. In particular, we propose an automatic framework for assessing and re-training failure prediction models on need-only basis, as manually re-training a predictor is a complex and high cost activity.

Chapter 6
A framework for continuous training of Failure Predictors

In the previous chapters we addressed the problem of training and assessing failure prediction models under the scarcity of failure data. We demonstrated that realistic software fault injection could be used for generating failure data in short time and proposed a framework for a sound assessment and comparison of alternative failure prediction models on a particular system installation. However, the scenario considered in such works is static, i.e., we assume that the target system does not change over time, which is a questionable assumption in some cases. Computer systems are nowadays expected to evolve, repeatedly and in several different ways. For instance, some hardware or software component may be changed during the target system lifecycle or the system itself may be subject to updates (e.g., a software patch), upgrades, or changes in its behavior (e.g., changing policies in memory management, installation of new protocols).

In scenarios where changes in the target system occur, the performance of failure prediction models may degrade (this is particularly evident in the case of *long-running* servers). Although one can develop adaptive failure prediction algorithms to cope with expected changes by adapting the prediction model to the environment, there is no guarantee that the expected changes are the only ones occurring (i.e., unexpected changes may also happen, degrading the prediction performance). Adapting the failure prediction model over time is thus a necessary step that requires additional effort, as training and optimizing predictors are still essentially manual procedures (e.g., (I. Irrera, Duraes, and Vieira 2014; I. Irrera et al. 2010; G.A. Hoffmann 2006; G. F. Hughes et al. 2002)), and failure data scarcity has an even greater impact (compared with a static scenario).

In this chapter we propose a preliminary **framework for the automatic adaptation of failure prediction models** (*Adaptive Failure Prediction Framework*, *AFP Framework*), which allows automatically re-training and deploying a failure prediction model when particular events occur, thus reducing the cost of the re-training process. The idea consists of having an integrated environment driven by configurable events that trigger the models' adaptation process. This is based on the use of the approach for generating realistic failure data proposed in Chapter 3 and virtualization to reduce the cost and impact on the target system. In practice, the framework has the following key characteristics:

- The framework supports **automated self-adaptation to accommodate evolving systems.** The failure predictors (re-)training process is automated, based on a modular event-driven architecture to detect when re-training is needed. An event corresponds to some occurrence in the system that may affect the prediction performance. The framework is configurable to meet the requirements and target system specificities, allowing the user to define events.
- The framework uses the approach proposed in Chapter 3 for **generating failure data based on software fault injection**, which reduces the time needed to (re-)train a predictor. This allows automating the failure prediction training, testing and validation, reducing time and human intervention, which is limited to the set-up of the framework.
- The framework uses a **virtualization environment to sandbox the fault injection process**, avoiding injecting faults in the target system (i.e., the fault injection takes place in a sand-boxed copy of the target system). In fact, as discussed in Chapter 4, injecting faults during operation would cause unacceptable side effects.

The *AFP Framework* is provided as a conceptual structure, including the architecture, the basic procedures involved (e.g., training, testing, fault injection and failure data collection, etc.), and guidelines for its implementation. Although being a preliminary work, the framework can already be implemented in a concrete target system by following the guidelines provided and by adapting the parts that are dependent on the specificities of that system (e.g., the workload to be executed on the target replica and the faults to inject). To demonstrate the effectiveness of the propose framework, this chapter includes a case study in which the framework is implemented in the context of a web server, showing that the solution is able to keep the predictor performance above a given threshold with small human intervention, under changing conditions.

Before describing the proposed framework, we also present a **study that confirms the need for adaptive failure prediction systems**. In particular, we study the performance of a failure predictor when used to forecast failures in an Apache Tomcat web-serving system under successive software updates.

The remainder of the chapter is organized as follows. Section 6.1 analyzes the performance of a failure prediction model in a changing system. Sections 6.2 and 6.3 present the proposed framework for implementing an Adaptive Failure Prediction environment. Section 6.4 presents the case study that demonstrates the applicability of the framework. Finally, section 6.5 concludes the chapter.

6.1 On the need for continuous training of failure prediction models

In this section we study the following hypothesis:

"a change in a system (e.g., a system upgrade) degrades the failure prediction performance, thus requiring a retraining of the prediction model".

We believe that this study is fundamental for motivating the framework proposed in this chapter, as training is a costly and complex operation (even considering the use of fault injection) and it should be done on a need-only basis. This way, understanding if changes effectively affect the failure prediction performance in order to avoid running a failure prediction system with degraded, possibly worthless, performance, and also to prevent unnecessary retraining efforts is a key aspect.

6.1.1 Overview of the study

The idea is to study the prediction performance looking for any degradation as the target system is updated. If some degradation is observed, then the predictor is updated (i.e., trained to predict failures in the updated environment) and re-assessed in order to understand if there is some performance recover. The assessment is based on the benchmarking approach presented in Chapter 5 applied to a virtualized target system (as before, virtualization provides a fast evaluation environment that isolates the target system for hosting fault injection).

Considering *B* and *C* as successive versions of an initial version *A*, and $P_{A,B}$, the performance (e.g., Precision, Recall, ROC-AUC, …) of the failure predictor trained with failure data from system version *A*, but being used in the system version *B* (updated system), we aim at confirming that:

$$(6.1) \qquad \begin{cases} P_{B,B} > P_{A,B} \\ P_{C,C} > P_{A,C} \\ P_{C,C} > P_{B,C} \end{cases}$$

The failure prediction model used is the Support Vector Machine (SVM) classifier with a *sliding window* and the purpose is to predict failures in a Windows OS server running the Java-based application server Apache, which is subjected to two

Table 6.1 - Failure predictor performance comparison.

		Testing on Tomcat version…		
		#A (6.0.36)	*#B (7.0.19)*	*#C (7.0.40)*
Training on Tomcat version…	*#A (6.0.36)*	$P_{A,A}$	$P_{A,B}$	$P_{A,C}$
	#B (7.0.19)	-	$P_{B,B}$	$P_{B,C}$
	#C (7.0.40)	-	-	$P_{C,C}$

updates. In practice, three versions of the Tomcat server (namely versions 6.0.36 (A), 7.0.19 (B), and 7.0.40 (C)) were used and the performance cases assessed are referred to as $P_{A,A}$, $P_{A,B}$, $P_{A,C}$, $P_{B,B}$, $P_{B,C}$, $P_{C,C}$ (see Table 6.1). The remaining cases were not analyzed, as we are interested only in the cases in which a predictor is trained with data coming from a previous version of the target system (and not the reverse).

To reduce the time needed for the experiments we installed the target system in three different virtual machines, thus parallelizing the collection of data and prediction results by a factor of three. The complete set-up is presented in Figure 6.1. The base version of Tomcat (6.0.36) was installed in the three virtualized systems, and in two of the virtual machines we updated Tomcat to version 7.0.19 and version 7.0.40, respectively, to implement the system upgrade/evolution aspect of our case study. The systems are installed on separated Citrix XEN servers (*hypervisors*) (Citrix) in order to avoid error propagation from one machine to another and potential influence in the variables monitored (it has not been proven yet that faults can or cannot propagate from a virtual machine to another). The controller systems are in charge of managing the experiments, analyzing the failure data coming from the target system, and hosting the failure predictor.

The characteristics of the different system are the following:

- **Sandbox system**: Intel i5-650@3.60GHz (quad-core) machine, 8GB RAM, 500 GB HDD, running XEN hypervisor server version 6.2.
- **Target systems**: 4 VCPUs, 4GB RAM, 50GB HDD, running Tomcat on the top of a Windows XP OS (SP3).
- **Controller systems**: Intel i5-650@3.60GHz machine, 8GB RAM, running a Windows XP OS (SP3).

Data coming from a target system S_x are organized in a dataset DS_x, containing Failure Data and Golden Data. As defined in Chapter 3, each dataset is divided in a Train dataset (TDS_x) and a Test dataset ($TTDS_x$), and the performance of the failure predictor when upgrading a system from version *A* to version *B* ($P_{A,B}$) is obtained by training the predictor using the training set TDS_A and then testing it using $TTDS_B$. As we must assure that the predictor performance does not depend on a particular dataset, the predictor is evaluated several times using *k-fold* cross validation.

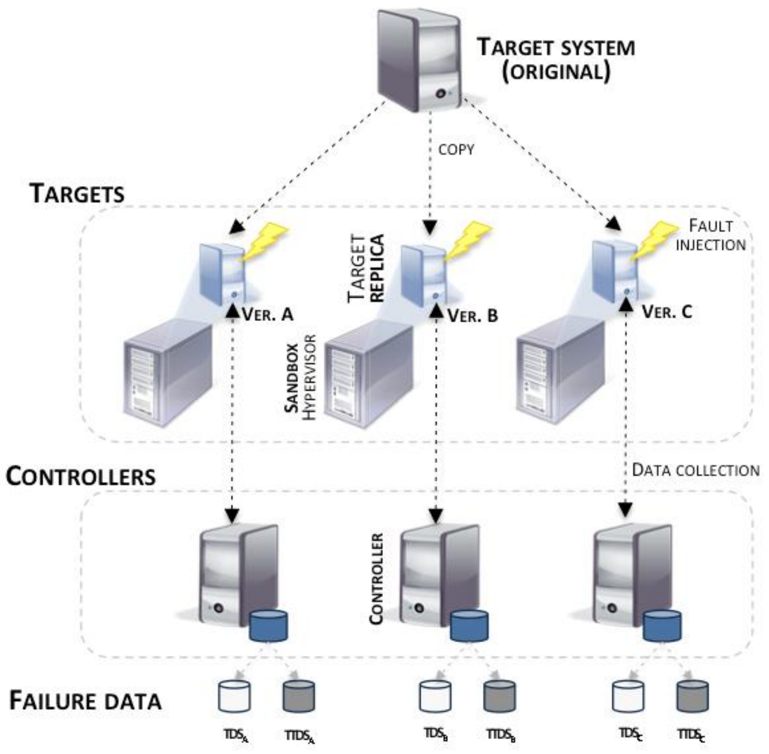


Figure 6.1 - Experimental setup parallelizing the generation of failure data.

The dataset DS_X is partitioned in k parts, obtaining k groups (folds) by taking different partitions as *training* and *testing* datasets each time, leading to different datasets DS_X^k. The predictor is then assessed using k different datasets $(TDS_X^k, TTDS_X^k)$, obtaining a distribution of the predictor performance $<P_{X,Y}^1, P_{X,Y}^2, \ldots, P_{X,Y}^k>$. We characterize the performance of the predictor in terms of the ROC-AUC.

The target systems are restored after each fault injection run, using the snapshotting and system restore functionalities provided by the XEN hypervisor. The approach proposed in Chapter 3 is used for generating the data, and the fault model, faultload, failure modes (*Crash* and *Hang*) and tools (including the SVM predictor with a *sliding window*) considered are the same used in the case study presented in Section 3.6. The difference is the workload, which consists in the set of operations defined by the TPC-W benchmark (Smith 2000), which is representative of real world-application scenarios (thus being a *realistic* workload).

TPC-W is a standard specification for benchmarking transactional web-serving systems, and the workload emulates typical operations executed by web servers in the form of an online bookstore. It includes an application implement as Java servlets to be deployed on the web server, and clients requiring the server to perform different types of operations. Hence, the TPC-W workload simulates an online bookstore serving client requests, where the clients are emulated by the controller system, and submit browsing and purchasing operations to the web server. In this

Table 6.2 - Monitored variables, an excerpt.

Variable name	Monitored component
Pool Paged Allocs	Memory
Pool Nonpaged Bytes	Memory
C2 Transitions/sec	Processor (core 0)
Page Faults/sec	Process java.exe
IO Read Bytes/sec	Process svchost.exe
Avg. Disk sec/Transfer	LogicalDisk
System Calls/sec	System
Semaphores	Objects
Context Switches/sec	Thread
Lazy Write Pages/sec	Cache

work we not consider workload variations, i.e., the workload parameters such as the client average request rate, the number of clients and the workload execution time, among others, are kept constant over time.

We monitored 233 numerical variables in each target machine whose values characterize the state of the OS resources, sampled at the rate of one value per second using the Logman tool that is included in Windows OSs family. As was done before, a three-step feature selection was used, reducing the set of variables to 25 (see more about this process in Section 3.6.2). The variable selection was performed for each tuple of values (Δt_l, Δt_p) on the system hosting the base version of Tomcat, for making the performance results comparable along the updates. An excerpt of 10 out of 25 variables is shown in Table 6.2.

6.1.2 Collected data

Details about the analysis parameters are shown in Table 6.3. The width of the *sliding window* is of 2 and 3 seconds, and the lead-time of the prediction Δt_l was between 10 and 40 seconds, with a prediction window Δt_p from 5 to 15 seconds. These values were chosen to simulate a realistic web-serving scenario. We believe that knowing if a failure is occurring 20 seconds in advance is sufficient for a system running Tomcat to save its status; obviously a larger prediction gives more time to react, but then the prediction performance may be poorer.

We executed about 2000 fault injection runs, each lasting about 240 seconds. One fault was injected in each run, leading approximately to 75 failures per target system (data about the failures observed in each Tomcat version are presented in Table 6.4). In practice, the workload was executed 2025 times (25 GR + 2000 FIR). The somewhat low number of failures was expectable, as there is no guarantee that the fault

Table 6.3 - Analysis parameters

Parameter	Values
Failure Modes	Crash, Hang
Workload	TPC-W
Run duration	240 s
Total runs	25 GRs + 2500 FIRs
Predictor	SVM (Gaussian kernel) + time windowing w= [2,3]
Variables (number)	170
Variable selection	Backward elimination + wrapper approach
Predictor Optimization	Grid search (gross) + Deepest descend (fine)
(γ, C) (Grid search)	γ= [2^{-10},1], C=[2^{-1}, 2^7]
Δt_l (Failure prediction)	10, 20, 30, 40 s
Δt_P (Failure prediction)	5, 10, 15 s
Results validation	5-folds cross validation

locations of the injected faults are within code actually executed (J. A. Duraes and Madeira 2006).

6.1.3 Results and discussion

Table 6.5 presents the average of the predictor performance in terms of ROC-AUC after a 5-fold cross validation (only w=2 is presented, as the results obtained with w=3 are not noticeably different), optimized according to the parameters (Δt_l, Δt_p), thus being the best performance results obtainable varying the couple (Δt_l, Δt_p).

Starting from the results in Table 6.5, we can notice that the average value of $P_{A,B}$ is smaller than the average of $P_{B,B}$, confirming the thesis that updating the system from version *A* to version *B* and *not* retraining the predictor may *cause* degradation in the predictor performance. When updating to version *C*, re-training the predictor is also the best choice ($P_{C,C}$=0.9889) if it was trained using version *B* ($P_{B,C}$=0.7689). However $P_{A,C}$ has the same value of $P_{C,C}$, possibly meaning that a re-train is not necessary. Nonetheless, such result may depend on several factors: for instance, version *A* and *C* of Tomcat may have similar behavior, even if a more likely reason is the fact that

Table 6.4 - Workload runs and failures occurred.

Tomcat ver.	Golden Runs	Fault Injection Runs	Failures detected
			System Hangs (%)
6.0.36 (A)	25	2500	73 (2.92 %)
7.0.19 (B)	25	2500	82 (3.28 %)
7.0.40 (C)	25	2500	79 (3.16%)

Table 6.5 - SVM performance comparison (sliding window w=2s).

		Testing on Tomcat version…		
		#A (6.0.36)	***#B (7.0.19)***	***#C (7.0.40)***
Training on Tomcat version…	***#A (6.0.36)***	0.9494	0.8543	0.9889
	#B (7.0.19)	-	0.9948	0.7689
	#C (7.0.40)	-	-	0.9889

such values are the average of the performance results coming from each fold, which may mask an existing difference between the two results.

A more detailed analysis of the predictor performance is presented in Figure 6.2, making use of Box-plots charts: each vertical bar represents the minimum, the maximum, the second and fourth quartile, and the median value of the ROC-AUC values obtained using the *k-fold* cross validation. We present the results using different windowing sizes (w=2 and w=3) to confirm that the results do not depend on w. When using obsolete training data, the performance $P_{A,B}$, $P_{A,C}$, $P_{B,C}$ drops for all values of w. However, in the case of $P_{A,C}$ the average performance seems not to get worse (seemingly, re-training is not needed) but the dispersion around the median value increases. This means that we have less assurance of obtaining a performance near a given value.

The $P_{A,C}$ case can be thus read as a loss of quality (performance) in the predictor behavior. It is worth mentioning that the performance $P_{A,C}$ is better than $P_{A,A}$. This result may be due to several reasons: most likely the Tomcat version 6.0.36 has some unpredictable behavior that version 7.0.x does not have, which makes the prediction harder. For instance, the Tomcat developers introduced in version 7 a "*Web application memory leak detection and prevention*" module (Vukotic and Goodwill 2011; "Tomcat Version 7 - Changelog"), which may be the cause for the data collected from version 7.0.x to contain less interference (due to better memory management).

Based on the results above, we can confirm the hypothesis that **upgrading a system from a version to a newer one may lead the prediction performance to degrade**, thus requiring retraining with data collected from the updated target system. Results also confirm that re-training can improve and recover the performance of the predictor.

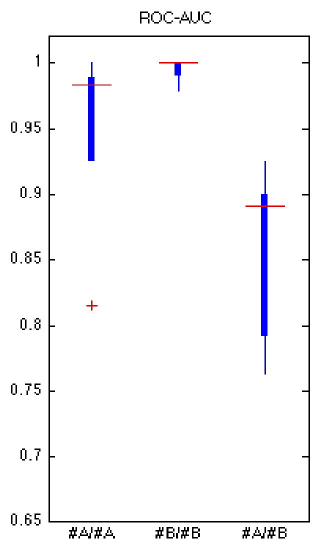
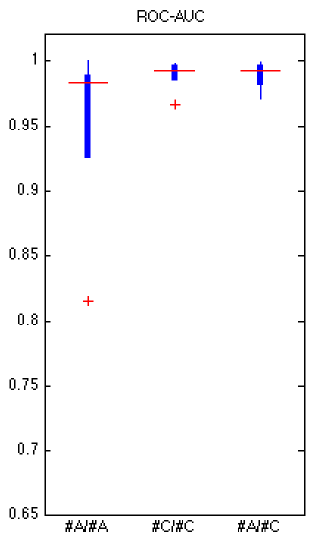
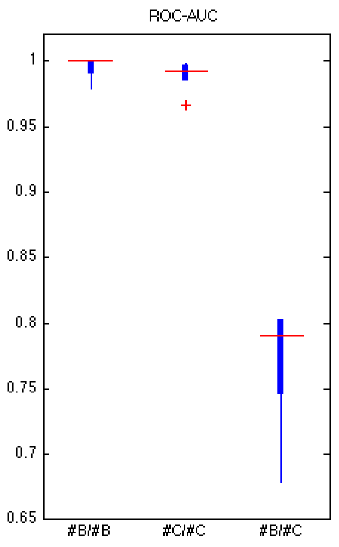
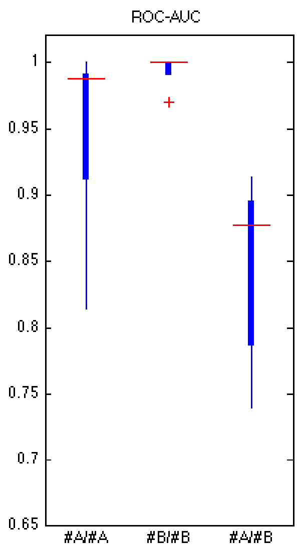
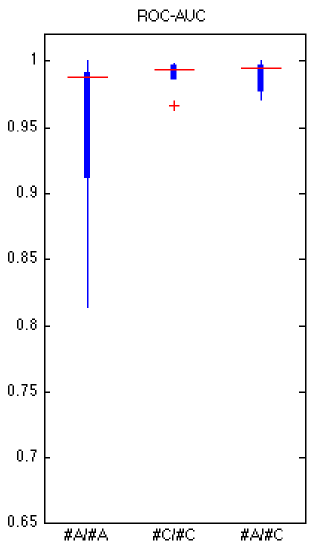
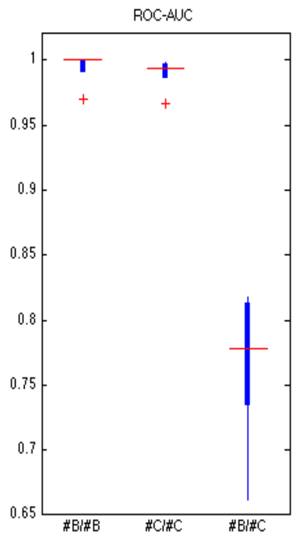
Tomcat web server versions X (older), Y (newer)		
A (6.0.36), B (7.0.19)	**A (6.0.36), C (7.0.40)**	**B (7.0.19), C (7.0.40)**
		
w=2	**w=2**	**w=2**
		
w=3	**w=3**	**w=3**

Figure 6.2 - SVM performance comparison (Box-plots)

6.2 AFP Framework concept and modules

The *AFP Framework* (Adaptive Failure Prediction Framework) aims at supporting the *automatic and event-triggered* assessment, re-training and deployment of failure prediction systems. The framework allows detecting when the performance of one or more failure predictors is below a given threshold and automatically retrains the predictors in such occasions. The ultimate goal of the framework is to provide the **support to maintain an optimal performance of failure predictors over time**, thus enabling the failure prediction to self-adapt to a dynamic (changing) target. The following are key characteristics of the AFP Framework:

- **Automation**: to reduce the need for user intervention to a minimum, the framework automatizes: *i)* the generation of failure data; *ii)* the assessment of failure predictor's performance using the collected data; *iii)* the training, testing and updating of the failure predictor model(s); and *iv)* the detection of update events. In particular, the framework implements the benchmarking approach proposed in Chapter 5 for the assessment of the failure predictors.
- **Configurable event-oriented architecture**: the framework is driven by an event-oriented logic. Events trigger the parts of the framework responsible for deciding if a retraining is necessary (and if so, to perform it). This logic avoids blind retraining, thus saving system resources and reducing costs. The events that are used by the AFP Framework can be configured by the user, which can also define specific reactions.
- **Sand-boxed, fault injection-assisted failure data collection**: the AFP Framework makes use of realistic software fault injection to cause realistic failures, by implementing the approach for generating failure data proposed in Chapter 3. A virtualized copy of the target system is used for generating failure data, thus preventing failures from impacting the target system, as discussed in Chapter 4. The original target system can be a virtualized or not-virtualized computer system, although the second case increases the usability of the proposed framework.

The AFP Framework is composed by several **modules**, each one with specific concerns. This facilitates the orchestration of the several operations needed and allows implementing each concern in an independent way. The framework core, presented in the next section, manages and organizes the use of the modules that compose the framework along several phases. In short, the modules of the AFP Framework are:

- **Sandboxing**: the AFP Framework makes use of virtualization as sandboxing solution for generating failure data. The sandbox manager is responsible for creating the target system replica and hosting it in a sandbox environment, besides being in charge of the target replica start, shutdown, reboot, and disposal operations. Recall that such operations are offered by several virtualization solutions, as for instance XEN and VMWare hypervisors (Citrix; Frappier 2014).

- **Failure data generation:** this module is responsible for the automatic generation of failure data by injecting faults in the target system replica as defined in Chapter 3, using realistic software fault injection and a realistic workload to exercise the copy of the target system. The collected failure data is organized as defined in Section 3.4. This module is also responsible for the detection of failures based on detectors provided by the framework user. Obviously, new detectors can be added after the framework deployment.
- **Failure prediction:** the main role of this module is to manage the failure prediction systems running on the target system, including conducting the predictors training and execution (runtime prediction). The module also makes use of the failure detectors (used for the failure data generation) for detecting failures in the target system. In practice, the failure prediction module makes use of data collected from the target system and from its replica. The data collected from the replica (Failure and Golden data) are used for training, while the data read from the target system during its execution is used at runtime to predict failures (the user has to define a buffer of data to perform the prediction (e.g., 10 seconds of data)).
- **Performance evaluation:** this module is responsible for the assessment of the failure predictor performance by implementing the benchmarking process presented in Chapter 5. In brief, the outputs of each failure predictor are collected and used to calculate a set of metrics chosen based on a given property (the AFP Framework adopts the recommendations for the choice of the metrics defined in Section 5.3). Evaluations are needed in several occasions (e.g., to train the failure prediction, to verify the need for a retraining).
- **Events manager:** this module is responsible for detecting events, triggering the re-training of the failure predictors, and checking if the predictor update is actually necessary. This module is customizable, as the events are implemented in a plug-and play manner, which permits the creation of new events (e.g., a timeout, the system being updated, the system configuration being changed, etc.) and new specific reactions, both at the framework set-up and after deployment. In practice, the manager detects the events that occur, starts the reaction relative to that event, and eventually re-trains the predictor. Afterwards, the module checks the eligibility of the requested update of the predictors by comparing the performance of the newly trained versions with the previous ones, updating the improved predictors only if there are potential performance gains.

The different modules of the framework are placed on the target and on the controller systems, as showed in Figure 6.3. The **user** is responsible of setting-up the AFP Framework, by conducting the following tasks:

- **Analyze the impact of alternative virtualization solutions on the failure data generation and select the most adequate one**, following the recommendations in Chapter 4.
- **Configure the process of cloning the target system into the sandbox environment**. If the target system is a virtualized one, then the user may take advantage of the native functionalities offered by the virtualization environment. On the other hand, if the target system is not virtualized, then the user must: *i)* create a virtual machine taking into consideration the hardware characteristics of the target system, in terms of CPU, memory, disk and networking, and *ii)* install the same software packages (including OS and services running on it) and applications (using the same versions running on the target system). Afterwards, user should create a backup of the virtualized target replica and configure the sandboxing module to restore such backup after each fault injection run. It is worth noting that, if the hardware or the software of the original system are updated, then that should be replicated in the replica, either by means of an automatic process or manually by the user.
- **Configure the failure data generation process based on the approach presented in Chapter 3**. This includes the definition of the workload to exercise the copy of the target system and the implementation of failure

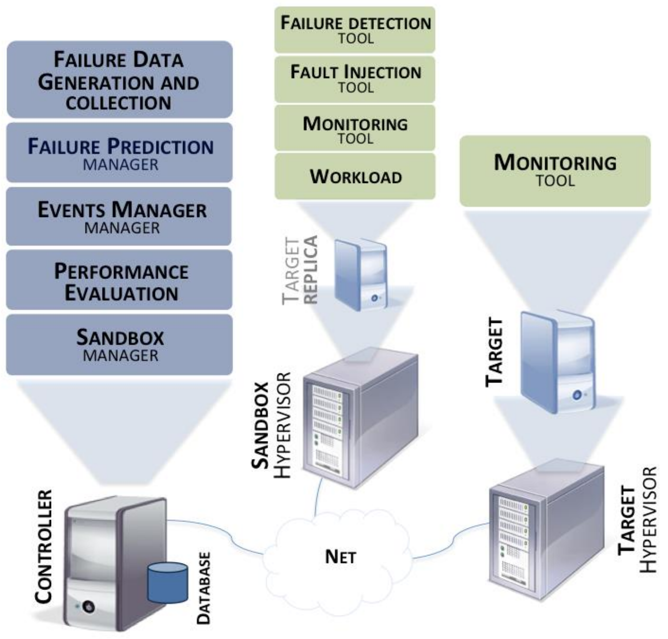


Figure 6.3 - The AFP Framework implementation.

detectors, which are fundamental pieces for the automation of the entire data generation process.

- **Configure the failure prediction models following to the recommendations presented in Chapter 3 and implement and configure the assessment environment for the failure predictors**. The proposal is to adapt the benchmarking framework proposed Chapter 5.
- **Define the events and the related reactions, considering the specificities of the scenario in which the framework is implemented** (e.g., the expected events representing evolution of the system). Both the event and its reaction must be implemented in a way that is compatible with the event manager module. In particular, the reaction must include information about the failure predictor re-training (e.g., train until a minimum false positive rate is reached) and about the failure data generation campaign (e.g., the number of golden and fault injection runs).

6.3 AFP Framework lifecycle and phases

The AFP Framework evolves along three phases: the preparation phase (in which a user must choose the failure prediction systems, the variables to monitor, etc.), the execution phase (failure prediction and event checking), and the training phase, performed several times during the framework execution. In each phase, the presented modules are orchestrated by the framework core, implementing the execution and training phases, with the goal of reacting and training the failure prediction systems when a specific event occurs. A schema is presented in Figure 6.4.

The **preparation phase** consists of a set of operations that the *framework user* must do in order to prepare the framework for execution, as presented in the previous section.

The **execution phase** consists of the effective prediction of failures and the continuous monitoring of the target system, checking for the occurrence of events for re-training the failure predictors, when necessary. In particular:

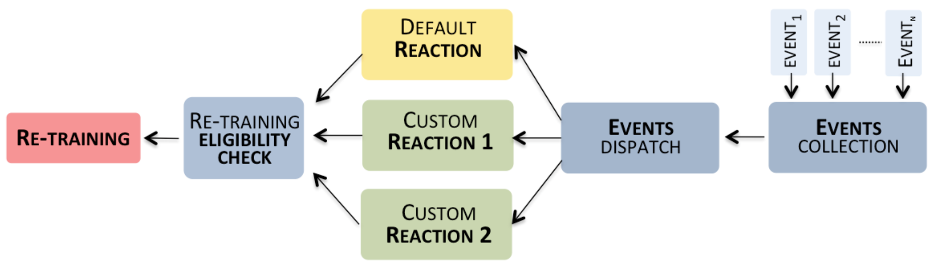


Figure 6.4 - Update events management and re-training execution.

- **Failure prediction**: the AFP Framework continuously collects data from the target system using a monitoring tool. Data are used by the failure predictor(s) to perform the prediction task.
- **Event checking**: the framework continuously checks if any new event occurs considered the ones defined. In the case an event is detected, the AFP Framework launches an update by calling a default procedure or a custom procedure associated to that event.
- **Failure predictor(s) (re-)training and update**: the re-training operation (or simply *training*, at the framework start-up) is executed on the occurrence of an event. The framework stops the execution of the events manager module, and starts the training phase, while the predictors continue in execution in the target system. Once the predictor is re-trained, its performance is compared with the one of the predictor still working in the target, which is then updated if the recently trained predictor performs better.

The **training phase** consists of training (or re-training) one or more failure predictors. The steps executed in this phase are:

1) **Target system replication**: the target system is cloned to a virtual machine, as configured by the framework user in the preparation phase. This step is repeated for each re-training, as the system replica must reflect the updated state of the target system.
2) **Data generation and collection**: executed in the context of the virtualized copy of the system, this is done several times to collect data from the replica system. Failure Data (FD) and Golden Data (GD) are collected, and datasets are built.
3) **Predictors training**: the failure prediction systems are trained using the dataset generated. The output of each predictor is collected for later analysis. This phase can be divided in three steps: i) training, where the failure prediction algorithms are trained using labeled data (*training data*) for discriminating *failing* from *non-failing* situations; ii) testing, where each failure predictor tries to label a set of unlabeled data (*testing data*); and iii) output collection, where the labels produced are collected.
4) **Metrics calculation and performance analysis**: the predictions from each algorithm are used to calculate a set of metrics, chosen based on the performance property that each predictor must address (e.g., high TP and low FP).

6.4 Case study: Adaptive Failure Prediction for a Tomcat web server

A case study was developed to demonstrate the process of implementing the AFP Framework and to analyze its efficacy in keeping the performance of the failure predictors above a certain threshold. The case study is inspired on the one presented in Section 6.1. The AFP Framework is implemented for continuously adapting a SVM classifier with the *sliding window* enhancement, predicting failures on a Windows XP OS machine running a Tomcat web server (the target system).

6.4.1 AFP Framework implementation

As shown in Figure 6.3, the modules of the framework are installed on a *controller machine* and remotely communicate with tools running on the *target system* and on its *replica*. Failure data and performance analysis results are stored in a *database*, managed by the controller system.

Citrix XEN hypervisors provide the sandboxing solution and host both the target replica and the original target system, thus simplifying the case study by allowing the target system's replication via a single machine migration operation, natively provided by the adopted hypervisor. The **sandbox manager** is implemented in the controller system and manages the hypervisors' operations. In our case, the sandbox manager is the client application for VMs management provided with the XEN servers.

Part of the **failure data generation module** runs in the controller system, namely a fault injection manager, a workload manager, and a module for communicating with the database that stores the failure data and the faultload. The Logman monitoring tool is installed both on the target system and on its replica for collecting failure and golden data, while the target system replica hosts the G-SWFIT fault injection tool.

The **failure prediction module** is implemented in the controller systems: it manages one or more failure prediction algorithms, using failure data stored in the database, and assesses their performance, when needed. The **event manager** is implemented on the controller system, which monitors the target system state for the defined events.

The **AFP Framework core** runs in the *controller system*. The framework modules are implemented using **PowerShell** and the **Microsoft Windows Management Instrumentation (WMI)** (both based on .NET Framework) (Siddaway 2012). The database used is a Microsoft SQL Server 2010.

The characteristics of the machines in which the AFP Framework was deployed are as follows:

- **Sandbox systems**: Intel i5-650@3.60GHz (quad-core), 8GB RAM, 500 GB HDD, running XEN server version 6.2.
- **Target system (and its replica)**: 4 VCPUs, 4GB RAM, 50GB HDD, running Tomcat on a Windows XP (SP3).
- **Controller system**: Intel i5-650@3.60GHz machine, 8GB RAM, running a Windows 7 OS (SP1).

6.4.2 Experimental campaign

For accelerating the experiments we installed two separated testbeds consisting of the same physical and software characteristics (i.e., each testbed made up by a separate controller, target and sandbox). In each testbed, the target system runs a Tomcat application server, which executes the workload of the TPC-W benchmark. The Apache Tomcat web server versions used are **5.5.36** (#**A**) and **6.0.2** (#**B**), and the intended performance analysis is summarized in Table 6.6. In practice, the framework should retrain the predictor as soon as the web server is updated from version #A to version #B, adapting it to the new target system configuration. Again, ROC-AUC is used for characterizing the failure prediction systems, whose goal is to maximize the recall and minimize the false positives.

The target system starts with a base version of Tomcat installed and then an update is performed. The events *"Tomcat update from version A to version B"* (U) and *"low prediction performance"* (LP) were defined. The first is triggered when Tomcat is updated, while the second is triggered when the ROC-AUC of the predictor falls below a certain threshold. The failure prediction module at the occurrence of each failure performs the verification of the ROC-AUC value against the threshold. In this way, we study the behavior of the predictors during golden runs and failure runs. Together with the events, we implemented the corresponding reaction, in which the framework re-trains the predictor, while the actual configuration one is left working.

In our experimental evaluation, we validate the trained failure predictors in the target system, where we inject faults to cause new failures. Of course, this is only done for experimental evaluation and validation purposes.

Table 6.6 - Failure predictor performance comparison

		Testing on Tomcat version…	
Training on Tomcat version…		***#A (5.5.36)***	***#B (6.0.2)***
	#A (5.5.36)	$P_{A,A}$	$P_{A,B}$
	#B (6.0.2)	-	$P_{B,B}$

The SVM failure predictor implementing the sliding window technique uses a window of 3 seconds. The choice of such value is based on the analysis of the results of the previous case studies: a SVM using a sliding window of width between 2 and 4 showed good performance improvements with a fair impact on the training cost.

The choices regarding failure modes to predict, failure detectors, faults to inject, workload, and variables to monitor are the same as in Section 6.1. The failure prediction lead-time Δt_l was between 10 and 40 seconds, with a prediction Δt_p of 5 or 15 seconds. These values were chosen to simulate a realistic web-serving scenario.

For the sake of simplicity, the TPC-W workload is executed on the target system and on its replica (when needed), hence the replica system uses a realistic workload for generating failure data (for details on such problem see Section 3.2.3). Both the replica system (for *training*) and the target system run the workload during a time period T of about 4 minutes.

6.4.3 Results and discussion

Table 6.7 presents an overview of all the scenarios considered in terms of the number of runs and o the observed failures. The target system executed the TPC-W workload between 500 and 800 times for each Tomcat version, resulting in a total of about 1600 runs for Testbed #1, and 1200 for Testbed #2. We observed a total of about 30 Hang failures in both targets, while Crash failures were observed only in the target running in Testbed #1 (7 crashes). We must highlight the fact that in this experimental campaign, Hang failures occurred also during Golden Runs, most

Table 6.7 - Runs and failures occurred.

		Testbeds					
		#1			**#2**		
Tomcat version	**Predictor status**	**Runs**	**Failures**		**Runs**	**Failures**	
			Crash	**Hang**		**Crash**	**Hang**
Ver. A *(5.5.36)*	*Before training*	*263 GR*	-	2	*250 GR*	-	-
		285 FIR	2	1	*250 FIR*	-	1
	After training	*165 GR*	-	-	*102 GR*	-	-
		127 FIR	-	5	*67 FIR*	-	8
TOT		*428 GR* *412 FIR*	*2*	*8*	*352 GR* *327 FIR*	*-*	*9*
Ver. B *(6.0.2)*			**Crash**	**Hang**		**Crash**	**Hang**
	Before training	*258 GR*	-	2	*100 GR*	-	1
		300 FIR	5	8	*250 FIR*	-	16
	After training	*117 GR*	-	-	*50 GR*	-	-
		100 FIR	-	6	*96 FIR*	-	9
TOT		*375 GR* *400 FIR*	*5*	*16*	*150 GR* *346 FIR*	*-*	*26*

likely due to residual faults, or to the workload we run. Despite not expected, we decided to use such events for training failure predictors together with the hang failures obtained from fault injection campaigns, as failure events may naturally occur also during golden runs.

Figure 6.5 presents the performance of the SVM predictor running on the original target system in terms of ROC-AUC, only for to the failure mode Hang, as the number of Crash failure events did not allow an extensive analysis of the predictor behavior. The x-axis represents the events observed on the *target* system, including failures. In particular, for each observed failure, the predictor labeled each data sample according to the parameters (Δt_l, Δt_p), and then the predictions were compared to the real failure occurrence time. The y-axis is the ROC-AUC value. For the sake of simplicity, we analyze the predictor performance fixing the parameters (Δt_l, Δt_p) to the values (10s, 10s). Such scenario represents the average behavior of the SVM predictor's performance observed in the present case study.

As mentioned before, the events considered in these experiments campaign were "*low prediction performance*" (LP) and "*Tomcat update from version A to version B*" (U). As shown in Figure 6.5 (a), the framework automatically reacted to the events LP and U, replicating the original system to the sandbox hypervisor, and retraining the predictor (R_1 and R_2) with data collected from the replica system. Retraining allowed the recovery of the prediction performance on both the testbeds. Each retrain was completely automatized and took about 3 days, mostly for collecting failure data, while training, testing and updating the failure predictors took few minutes. Such results confirm our expectations, as the average time we experienced in re-training failure predictors in the case studies presented in the previous chapters took much longer (about 10 days, in average), where most of the time was spent to organize the system replication, the data organization and performance results analysis.

Figure 6.5 (b) shows similar results: starting from the LP event, the retraining R1 permitted the SVM predictor to achieve a performance greater than the threshold value of ROC-AUC=0.8. The update of the Tomcat web server from version A to version B caused a degradation of the performance, which only after two failures resulted in a "*low prediction performance*" LP event. The retraining event R_2 enabled the predictor to restore its prediction performance above the defined threshold. The re-training R_2 after the update event U took about 6 days on Testbed #2 took. This was due to the SVM optimization algorithms, probably caused by the fact that data coming from Testbed #2 were noisier than data collected from Testbed #1 (this may be due to varying environment conditions, as for example the dynamic workload here used, or differences between the Testbed #1 and Testbed #2 system replicas). However, still in this case, automatizing the re-training process is convenient.

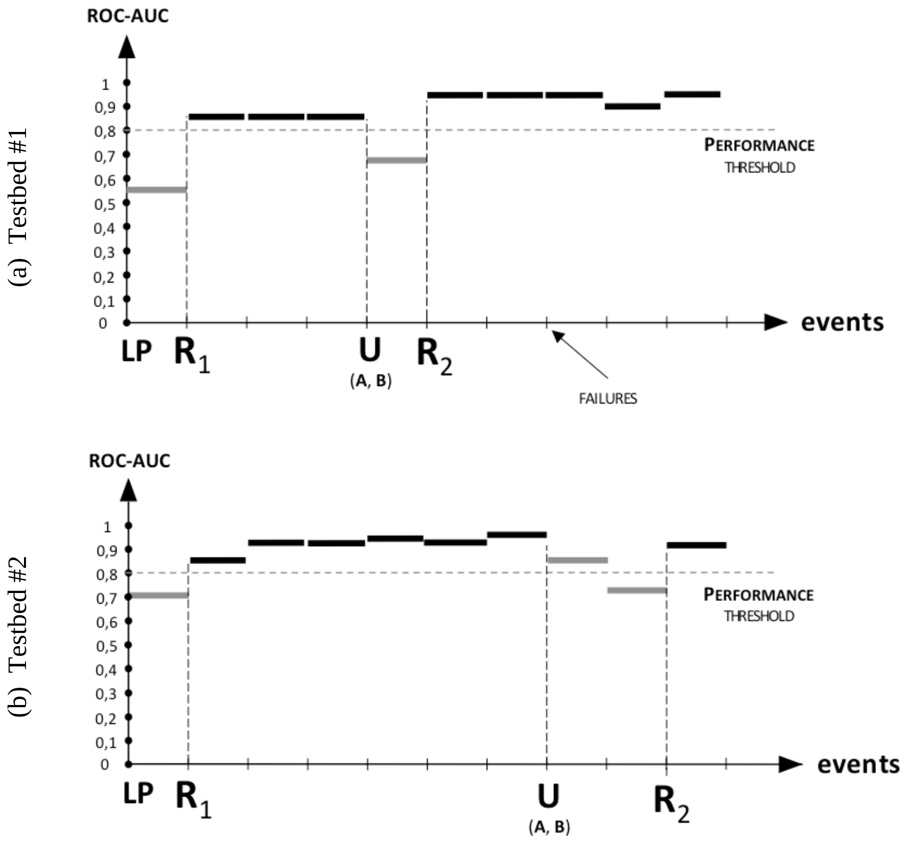


Figure 6.5 - The predictor ROC-AUC predicting *hang* failures, using parameters (Δt_l,Δt_p)=(10s, 10s)

Results confirm that, once configured, the AFP Framework was able to react to defined events and bring the predictor performance to an optimal working value, generating failure data in short time (between 3 and 6 days) without human intervention. This suggests that the framework can be used for continuously retraining failure predictors.

6.5 Final remarks

In this chapter we addressed the problem of self-adapting failure prediction models in the context of dynamic computer systems, by proposing a preliminary event-driven, user-configurable and modular framework, called *AFP Framework*. Such framework uses virtualization as a sandboxing solution for generating failure data (when needed) by injecting software faults into a replica of the target system using the approach proposed in Chapter 3. The need for implementing a framework for automatically re-training failure prediction models was studied in a case study.

The proposed framework includes several modules that a user can easily implement in a specific environment, including the sandboxing module, the failure data generation module, the failure prediction and performance evaluation modules, and the events management module. We presented an implementation of the AFP Framework in a specific case study: an SVM-based failure predictor protecting an Apache Tomcat web server running on a virtualized Windows XP. The results obtained in the case study demonstrate that the framework implementation was able to keep the predictor performance above a threshold across updates in the target system, in a small amount of time and with reduced human intervention.

Chapter 7
Feature Selection based on symptoms identification: Case Study

Choosing a set of variables (or **features**) for modeling a particular process or event is a key problem, as such set must characterize the process or event in a complete way and without redundancy. A given set of variables is optimal if the obtained model can optimally predict the process or event, i.e., the model performance is maximized. In particular, including *uninformative* or *weakly informative* variables in the model may result in a worthless increase of the complexity, also resulting in an increase of the time needed for model building (training time). On the other hand, the use of *wrongly informative* variables affects the prediction performance of the model (H. Liu and Yu 2005).

Several works have been proposed so far for addressing the feature selection problem, divided into two approaches: filter and wrapper (see Section 2.2.5). However, in the particular context of failure prediction, the existing approaches do not take advantage of information regarding the failure occurrence events, which can obviously result in limitations in the quality of the failure prediction models. On one hand *filter approaches* only analyze the correlation between variables, excluding the dependent ones. On the other hand, *wrapper approaches* perform a much more complete variables analysis, including selecting a set of variables, building a model with such set and repeating the procedure until an optimal model is obtained. However, albeit representing an optimal approach, it requires a huge use of computational resources and a long optimization time, making a complete selection process unfeasible.

In the particular context of failure prediction, the type of information that constitutes failure data, together with the coding of those data (e.g., using integer instead of real values, normalize the numerical values), influences the prediction quality of a model (G.A. Hoffmann, Trivedi, and Malek 2007). As shown by (G.A. Hoffmann, Trivedi, and Malek 2007), the variables chosen by experts are likely not to be the optimal set for prediction, and a proper feature selection procedure is needed. Feature selection techniques can be applied to the failure prediction problem and an improvement of the classical *filter/wrapper* feature selection schema was proposed by (G.A. Hoffmann, Trivedi, and Malek 2007) in the form of a *probabilistic wrapper approach* that makes use of information about the correlation of a set of variables with the target (i.e., the performance of a generic prediction model) in a probabilistic manner. The problem is that such approach makes use of a generic prediction model that was implemented by the same authors (the Universal Basis Function, UBF (G.A. Hoffmann, Trivedi, and Malek 2007)), which narrows the results to that particular predictor.

In this chapter we present a case study where we analyze the effectiveness of the failure symptoms identification method proposed in Chapter 4 in addressing the feature selection problem. The idea is to use that approach for correlating the symptoms presented by each variable with the occurred failures, and rank the variables according to their correlation, using the technique proposed in Chapter 3 for generating the failure data. We believe that the information about the symptoms of failures that each monitored variable shows can indicate the most adequate variables for being used for failure prediction.

It is important to emphasize that the case study allows a preliminary analysis of the applicability of a symptoms-based feature selection technique for the failure prediction scenario. We believe that such feature selection approach can help in the selection of variables independently from the failure prediction model being used, and that it can also be applied as a complementary technique to improve the blind selection of the *filter* approaches and/or to decrease the complexity of *wrapper* approaches.

The outline of the chapter is as follows. Section 7.1 recalls the feature selection approach and overviews the case study and Section 7.2 presents the experimental campaign. Results are presented and discussed in Section 7.3. Finally, Section 7.4 concludes the chapter.

7.1 Feature selection approach and study overview

The failure symptoms identification approach proposed in Chapter 4 is here adapted for selecting the best variables for predicting failures. Specifically, it is divided in three phases:

1) **Generating Golden Data and Failure data** from the target system using the approach for failure data generation based on realistic software fault injection presented in Chapter 3.
2) **Identifying failure symptoms and their correlation with the observed failures**, in the following way:

    a) The **failure symptoms are identified using the anomaly detection-based method** proposed in Chapter 4. In practice, a normal behavior profile is built for each variable using Golden Data. The behavior of that variable is compared with the profile during Failure Runs and other Golden Runs. The variable presents a symptom if its behavior differs from its nominal profile, according to specific rules.

    b) The **symptoms shown by a single variable are correlated with the failures** (and no-failure events) observed on the target system according to a specific metric. A variable is correlated to a failure if it presented a symptom when the failure occurred and did not present any symptom during Golden Runs. Correlation can be based in different metrics, as for instance metrics based on the contingency table (see Section 2.2.3). In this case study we adopt the F-Measure.

3) **Rank the monitored variables according to their correlation values**. The variables are ranked based on the correlation between the symptoms identified and the observed failures. Different ranks can be obtained in different scenarios, as different environmental parameters (e.g., workloads) do influence failure data. The variables that show the highest rate of valid symptoms have the highest likelihood to enable an optimal failure prediction.

As in Chapter 4, the setup for the experiments includes a monitored target system (system for which we want to identify the best failure prediction variables) and a controller system (in charge of controlling the execution of the experiments), installed in different machines (to isolate the effects of the experimental control from the monitored system) connected using a dedicated network (to avoid interference from external network traffic).

We conducted a variable selection campaign for a Windows XP OS-based target machine. The **experimental setup** consisted of two key elements (see Figure 7.1): the monitored system, on which the faults were injected, and a driver system (or *controller*) for controlling the experiments and collect, archive, and analyze monitored data. Both the monitored system and the driver system consisted of a machine with a Pentium IV HT 3GHz processor, 2GB of RAM, and a 200GB SATA

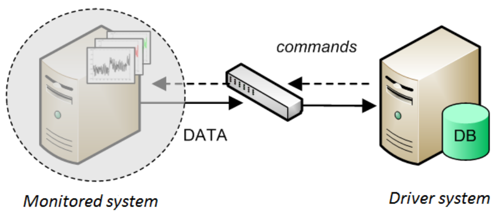


Figure 7.1 - Experimental setup

hard disk, running Windows XP (SP3) Operating System. The two machines were connected via a Fast Ethernet network.

Two workloads have been used in the experiments to assess if different operational profiles may lead to different failure prediction symptoms, as these two workloads stress the system in different ways. The workloads are the same used in the previous chapters: a light workload based on the 7-Zip application (WKL_1) and a heavier workload based on the COSBI OpenSourceMark benchmark suite (WKL_2) (see details in Section 4.6).

As before, the LogMan tool was used for monitoring data at the maximum sample rate of 1 value per second. The starting set included 387 variables selected manually, describing the state of the operating system resources, the state of the processes running, the availability and usage of network related resources, and information on terminal and disk I/O activity. Note that we did put some care on the selection step and tried not to exclude potentially good parameters (in case of doubt, we considered the parameter for monitoring).

The choices taken about the **failure modes** and **faults to inject** are the same as in Chapter 4. The difference is that, besides *Crash* and *Hang* failures, in the present case study we also consider the failure mode *Incorrect Results*. In the case of WKL_1 such failure is detected by checking a checksum, while for WKL_2 a failure is detected when the benchmarking results are out of a nominal range computed during the Golden Runs. For *Crash* and *Hang*, the failure detectors are the ones already used before. The G-SWFIT tool was used for injecting software faults in the dynamic library *kernel32.dll* following the recommendations by (J. A. Duraes and Madeira 2006).

For the symptoms identification, we computed the F-Measure, addressing the maximization of the true positives (a symptom corresponds to a failure) and minimizing the false positives (a symptom is identified but no failure occurred) and the false negatives (the variable showed no symptom, but a failure occurred). The higher the predictive power of the variable, the higher is the F-Measure (which obviously ranges from 0 to 1). Obviously, different ranks can also be obtained using the individual *Precision* and *Recall* measures, as well as many other metrics. The tolerance for the **bounds of the model** representing the typical behavior of each parameter was of 10% (tuned based on the analysis of the experimental results).

7.2 Experimental campaign

Table 7.1 presents the overall characterization of the experiments. A total of 1100 golden runs and 1143 fault injection runs were conducted. The duration of each run was of 600 seconds, leading to an experimental campaign of 16 days (around 1,345,000 data points). In each fault injection run, software faults were injected approximately 70 seconds after starting the execution of the workload (value defined based on the analysis of the ramp up time of the tested configurations). The number of injected faults for each run ranged from 1 to 5. Differently from the case studies in the previous chapters, here we injected more faults to increase the failure rate occurrence, as we are not particularly concerned with the accuracy of the generated data, but with the analysis of the correlation of variables with the failures. Failures were observed in a subset of the fault injection runs (111 for the configuration using WKL_1 and 98 for configuration using WKL_2).

The most predominant failure mode observed was the system *Hang* and the least frequent was the generation of *Incorrect Results*. This shows that in most failure situations the faults injected leaded the OS to block and that the propagation of errors to the application level was minimal. It is also worth noting that, differently from the percentage of *Hang* faults observed in the experimental campaigns presented in the previous chapters (in 2% of the runs, in average), the percentage of *Hang* failures is now of 15%. This is due to the fact that in the present campaign we injected a higher number of faults per run (between 1 and 5, as mentioned before).

7.3 Results and discussion

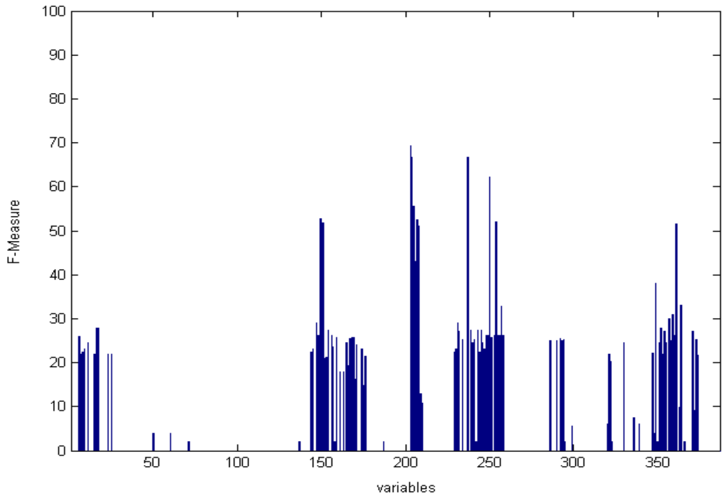
Figure 7.2 shows the F-Measure for the 387 parameters monitored in both configurations, computed considering all the failures observed. Only a small number of parameters present an F-Measure greater than zero (77 and 109 for the configurations running WKL_1 and WKL_2, respectively), which suggests that most of the monitored parameters are not useful for supporting failure prediction. A small

Table 7.1 - Overall characterization of the experiments

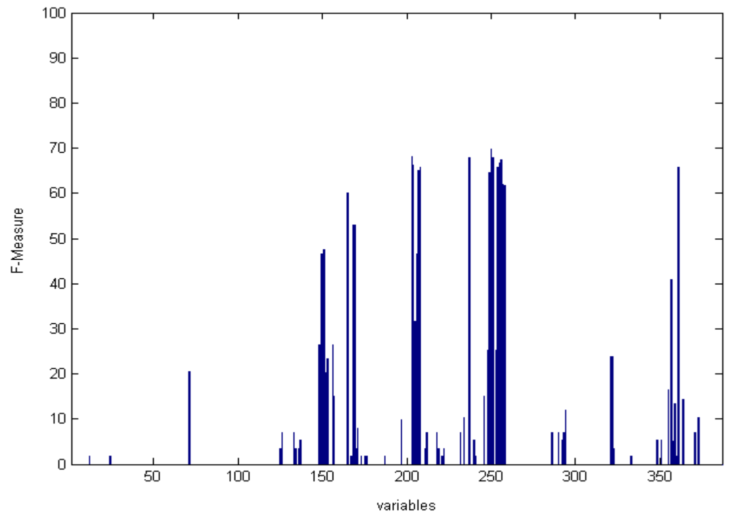
Workload	# Golden Runs	# Fault Injection Runs	Failures detected			
			Total %	*Incorrect Results %*	*System Crash %*	*System Hang %*
WKL_1	500	500	22,20% (111)	0	4% (20)	18,20% (91)
WKL_2	600	643	15,24% (98)	1,87% (12)	3,58% (23)	10,58% (63)

subset of the monitored variables shows a positive correlation with the occurred failures, presenting an F-Measure higher than 50% (17 in WKL_1 and 12 in WKL_2).

Table 7.2 presents the Top-10 parameters ranked with respect to *Hang* failures for both workloads. The results for the *Crash* and *Incorrect Results* are omitted due to the low number of such failures observed (that would make the analysis not precise). The top-variables vary depending on the system configuration, which is expectable. However, there are five parameters that show up in both cases, with quite similar F-Measure (rows in gray). Although this confirms that the predictive value of the parameters might be influenced by the operational profile of the system, it also suggests that there may be a small set of parameters whose predictive power is quite independent of the configuration.



(a)



(b)

Figure 7.2 - F-Measure of the 387 parameters monitored in both configurations, relative to all the failures occurred (Crash, Hang and Incorrect Results).

Table 7.2 - Top-10 parameters, according to F-Measure (in percentage)

Workload #1, *Hang*		
Parameter	***Parameter Name***	***F-Measure***
250	Pool Nonpaged Bytes	70.42
203	Events	68.57
237	Handle Count	68.12
251	Pool Paged Bytes	68.09
255	Virtual Bytes	67.67
256	Virtual Bytes Peak	67.63
204	Mutexes	66.18
208	Threads (Objects)	65.69
254	Thread Count	65.69
361	Threads (System)	65.69
Workload #2, *Hang*		
Parameter	***Parameter Name***	***F-Measure***
203	Events	73.79
237	Handle Count	72.55
204	Mutexes	67.37
250	Pool Non-paged bytes	60.87
205	Processes	54.55
149	Memory Avail. Bytes	51.09
150	Memory Available KB	51.09
151	Memory Available MB	50
254	Thread Count	49.54
207	Semaphores	46.62

Table 7.3 shows the detailed results for the Top-10 variables presented in Table 7.2, thus relative to the <*WKL_1, Hang*> and <*WKL_2, Hang*> scenarios. As shown, precision is quite high for WKL_1 (always above 90%), but significantly lower for WKL_2 (less than 90% in six cases). On the other hand, recall is quite low in both configurations, suggesting that there were a large number of cases in which variables were not able to, individually, show any symptoms.

Another result that is worth highlighting is the number of false positives and false negatives of the top-10 variables. In the <*WKL_1, Hang*> scenario (Table 7.3 (a)) the variables show a high precision of the symptoms to address the occurred failures (almost only true positives, with very few false positives). On the other side, the same variables miss some of the occurred failures (false negatives), having thus a low Recall. In the <*WKL_2, Hang*> scenario (Table 7.3 (b)), on the other hand, only six out of ten variables had the same behavior presented by the variables in Table 7.3 (a), while the tendency is to present more false positives, resulting in variables with a lower Precision but with the same Recall values. This may be due to the fact that WKL_2 is more complex than WKL_1, introducing dynamics in the data that the threshold model proposed in Chapter 4 may be not able to recognize. The results also highlight the fact that the variables 203 (Events), 237 (Handle count) and 204 (Mutexes) presented the same behavior (low false positives and negatives) in both the scenarios, thus emphasizing the existence of a set of variables that can be used independently from the workload. It is also worth noticing that, given a couple of variables, they may recognize different failure events, thus a combination of such variables may provide an optimal set for predicting failures.

The analysis above confirms that the symptoms identification-based approach can be used as a feature selection technique. The approach allowed analyzing the symptoms shown by single variables in different scenarios in about two weeks (16 days), being effective in helping restricting the set of variables that could be further analyzed by *filter* or *wrapper* feature selection approaches. In practice, it enables one to have insights on variables and their likelihood in predicting failures. Moreover, our approach presents the advantages of both filter and a wrapper approaches, as the analysis of the failure-related data is done without the use of any failure predictor (as *filter* approaches), while the symptoms correlation is based on failure prediction performance metrics (similarly to a *wrapper* approach, in which the performance value of a predictor is used in the selection).

Table 7.3 - Detailed results the Top-10 parameters (*hangs* only)

Parameter	250	203	237	251	255	256	204	208	254	361
TP	50	48	47	48	46	47	45	45	45	45
FP	1	1	0	2	1	1	0	1	1	1
FN	41	43	44	43	45	44	46	46	46	46
TN	388	388	389	387	388	388	389	388	388	388
Precision (%)	98.04	97.96	100	96	97.87	97.92	100	97.83	97.83	97.83
Recall (%)	54.95	52.75	51.65	52.75	50.55	51.65	49.45	49.45	49.45	49.45
F-Measure (%)	*70.42*	*68.57*	*68.12*	*68.09*	*66.67*	*67.63*	*66.18*	*65.69*	*65.69*	*65.69*

(a) **Workload #1,** Total number of *hang* failures = **91**

Parameter	203	237	204	250	205	149	150	151	254	207
TP	38	37	32	35	24	35	35	23	27	31
FP	2	2	0	17	1	39	39	6	19	39
FN	25	26	31	28	39	28	28	40	36	32
TN	543	543	545	528	544	506	506	539	526	506
Precision (%)	95	94.87	100	67.31	96.00	47.30	47.30	79.31	58.70	44.29
Recall (%)	60.32	58.73	50.79	55.56	38.10	55.56	55.56	36.51	42.86	49.21
F-Measure (%)	*73.79*	*72.55*	*67.37*	*60.87*	*54.55*	*51.09*	*51.09*	*50*	*49.54*	*46.62*

(b) **Workload #2** Total number of *hang* failures = **63**

7.4 Final remarks

In this chapter we presented a case study on the effectiveness of the failure symptoms identification method proposed in Chapter 4 in addressing the feature selection problem. The approach followed allows ranking a set of variables according to the symptoms each variable shows at the occurrence of a failure. The symptoms are correlated with the failures using the F-Measure.

The symptoms identification-based approach is adapted in three phases, including a failure data generation phase, symptoms identification and correlation phase, and a variables ranking phase, intended to identify the features that are mostly correlated with failure events.

We studied the effectiveness of the approach by running a campaign for analyzing variables on a Windows-based system, running two different workloads. We collected 387 variables, representing the properties and status of the operating system and target system's hardware, along a 16-days fault injection campaign, and injected a total of 1143 faults, collecting 209 failures divided into *crash*, *hang* and *incorrect results*. Results show that the proposed approach is quite effective and easy to use for identifying the parameters that show a good correlation with failures, allowing **narrowing the focus on small sets of variables** that present a positive correlation with the occurred failures.

Chapter 8
Conclusions and Future Work

This thesis proposed methodologies to advance the state-of-the-art on failure prediction by making use of injection of realistic software faults to support the generation of failure data that can be used for training, assessing and comparing failure prediction models on a particular system installation.

The thesis started by proposing a **framework for generating failure-related data to be used for training and testing failure prediction models in a short time**, **based on the injection of realistic software faults**, which accelerate the occurrence of failures. The framework should be implemented on specific target systems, for collecting extensive and realistic datasets that take into account the characteristics of the environment, and encompasses all the steps necessary for the generation of the failure data, including the definition of the types of faults to inject, the identification of the workload to be executed by the target system, the selection of variables representing the behavior of the system, the detection of failure events, and the data generation, collection and the dataset building process. The data generation framework also includes an approach for assessing the accuracy of the generated failure-related data, allowing increasing the confidence in using such data and enabling a controlled and quality-driven generation process.

A case study was devised to demonstrate and validate the proposed approach in assessing the performance of a **novel online failure prediction model** that improves the failure prediction quality of a generic classifier-type predictor by including the time dimension in the prediction task. The case study uses a Windows-based software fault injection tool implemented at University of Coimbra following the G-SWFIT recommendations. The proposed method allowed the analysis of an SVM-based implementation of the failure prediction model running in a Windows XP OS environment in four different scenarios, considering two different failure modes and two different workloads. Results showed the effectiveness of the proposed prediction model and confirmed that failure data could be generated by software fault injection.

A solution to the problem of generating failure data on computer systems after deployment was proposed. Such solution is based on the use of **virtualization as a sandboxing environment for generating failure-related data**, hosting a copy of the target system. A virtualized copy of a system is easily manageable and easily recoverable, which eases the data generation process in what concerns the removal of the injected faults and of possible damages caused by their activation. We presented a solution for studying the applicability of such solution based on the concept of failure symptoms identification and correlation with failures. The approach was used to study the correlation of failure data generated from a system with data collected from several virtualized copies. In practice, we analyzed the impact of the virtualization layer of four different commercial hypervisors in four scenarios (two different workloads and two different failure types) running a Windows XP OS environment, showing that small sets of variables show similar symptoms in the original system and in its virtualized copies, although other variables differ across different scenarios and across the different hypervisors. Such results allowed confirming that virtualization can be used as a solution for generating failure data, although an assessment of the variables in common between the target system and its virtualized copy is needed.

Given the need for a fair and sound assessment of alternative failure prediction models in the context of a specific target system, we proposed a conceptual **framework for implementing benchmarks for failure prediction models (**Failure Prediction Benchmark or *FP Benchmark*). The framework envisages the necessary steps to implement the benchmarking process, including the definition of the faults to be injected for generating the failure data, the metrics that must be used for assessing and comparing the different solutions, the characteristics of the workload, among others. We practically demonstrated the effectiveness and applicability of a concrete Failure Prediction Benchmark in assessing and comparing alternative SVM-based failure prediction algorithms in a Windows XP OS environment. Two different workloads and two failure modes were considered, and four failure predictors were assessed and compared in four different scenarios. Results showed that the procedure could be used for training, testing, and comparing failure predictors in a cost-effective and easy way. The study of the failure prediction results and the validation of the benchmark properties suggest that the proposed benchmark can be used in the field.

A **conceptual framework for the automatic and continuous self-adaptation of failure prediction systems** (Adaptive Failure Prediction Framework, or AFP Framework) was also proposed. The goal is to train failure prediction models at runtime on the occurrence of specific events (e.g., a software update), collecting failure data when needed by using our approach for generating failure data, and using virtualization as a sandboxing environment for performing the fault injection process, taking into account the impact of the virtualization on the data generation. The training process is automated and based on a modular event-driven architecture to detect when re-training of predictors is needed. A concrete implementation of the AFP Framework was applied in a specific case study including an SVM-based failure predictor applied to an Apache Tomcat web server running on a virtualized Windows XP. Although preliminary, the framework was able to automatically perform training and testing activities for the predictor to be at a nominal performance, despite updates in the Apache Tomcat software, within a small amount of time and human intervention.

Finally, we studied the use of the symptoms identification approach (proposed for analyzing the impact of virtualization environment) as a **method for selecting the best system variables to be used for predicting failures**, including ranking a set of variables according to the correlation between the symptoms showed and the failure events. A case study envisaging a campaign for selecting variables for a Windows XP OS-based system, running two different workloads, was developed. The approach effectively allows the selection and ranking of a set of variables positively correlated with the observed failures. Results also showed that different rankings are obtained in different scenarios, thus confirming the influence of the workload on failure data as found in the previous case studies, but also highlighting some variables in the top of the rankings that can be considered independently from the workload.

The work presented in this thesis contributes towards moving forward the state-of-the-art in the following ways:

- The injection of realistic software faults to generate failure-related data allows overcoming the limitations of using existing failure data repositories (hosting failure data from several kinds of systems). In fact, such solutions clearly limit the optimal modeling of the predictor, as a target system may evolve over time (leading to the need for new failure data to be collected), and the failure data may come from different systems, or even different configurations of the same type of system. In particular, our results show that even the use of a different workload may impact on the optimal performance of a failure predictor. Moreover, the failure data accuracy analysis is a novel approach in the failure prediction context.
- The proposal of using virtualization environments as sandbox solutions for generating failure data is novel in the field of failure prediction, as well as the approach proposed for assessing the impact of virtualization layer on the failure data.
- The framework for benchmarking failure prediction models is the first one in this direction. Although the availability of failure data repositories can provide datasets for the assessment and comparison of failure prediction models, using such datasets is not sufficient for conducting a fair and sound

comparison: the assessment of failure prediction models with failure data collected from several systems does not allow taking into account the behavior of the target system on which the predictors will run. A concrete benchmark implemented on the system that will host the failure predictor supports such fair and sound analysis.

- The framework for adaptive failure prediction is the first work addressing automatic and event-based automation of failure prediction models adaptation over time. Few works, often resulting in manual approaches, have addressed the problem of adapting failure prediction models along the evolution of a target system.
- The adaptation of the symptoms identification-based approach to the feature selection problem permits an *a priori* analysis of the variables based on the failure symptoms they show and the correlation of such symptoms with effectively occurred failures. Such information gives the user the possibility to drastically reduce the number of variables to select for failure prediction, with the possibility to be integrated with complementary feature selection steps, based either on filtering or wrapping.

Future work

The work presented in this thesis has contributed to gain a broad experience on the challenges to be addressed when using fault injection to improve failure prediction. The light shed by our research helped identifying the following research directions to pursue:

1. **Validate synthetic data against real-world failure data and improve the data generation process.** As presented in Chapter 3, the generation of failure data is influenced by several factors, and validation is needed for assuring the training and testing of failure predictors to be accurate. Analyzing real failure data is needed for improving the generation process, including the selection of the most suitable set of metrics for estimating the accuracy of synthetic data with respect to real data, and the definition of the workload and the faultload used, among others.
2. **Implement Adaptive Failure Prediction Frameworks for specific classes of systems.** The AFP Framework proposed in Chapter 6 must be further specified to address the specificities of different computing environments. The challenges to address include the identification of classes of systems that can host an AFP framework, the replication of the target system and its workload, and the automation of such activities.
3. **Study alternative sandboxing solutions to the use of virtualization for generating failure data by injecting faults in a target system.** The works in

Chapter 4, Chapter 5 and Chapter 6 allowed us to identify the limitations in using virtualized environment, in particular their impact on the generated failure data and the difficulty in using that solution in complex systems.

4. **Propose a feature selection algorithm based on the symptoms identification approach,** proposed in Chapter 7. In particular, one should study the impact of using combinations of variables for predicting failures.
5. **Implement fault-injection-enhanced failure prediction to mission- and safety-critical systems.** We believe that this is the very next step to be pursued, namely investigating the properties that an *enhanced* failure prediction environment must encompass in order to address the requirements of such type of systems.

Several research topics are currently scheduled as a continuation of the work presented in this thesis, in particular:

1. **Implement fault-injection enhanced failure prediction OTS platform for predicting failures of software systems**. The aim is to provide an off-the-shelf platform hosting several failure prediction algorithms, integrated with fault injection tools, to serve as a component in complex failure prediction environments.
2. **Validate the impact of the workload on the failure data and failure prediction performance by using more complex workloads.** A first idea is to set up an experiment in which an SVM-based failure predictor is used on an Apache Tomcat web server running several configurations of the TPC-W workload. The goal is to mine relations between workload profiles in terms of <CPU, memory, I/O> dimensions and the effects on the failure prediction models.
3. **Study the *predictability* property of different failure types** based on the concept of *warning time* Δt_w in Salfner's failure prediction model. The reasoning is that some failure events may not be predictable at runtime and that fault injection may be useful for investigating the classes of failures that are predictable in a minimum time Δt_w. This may help in the definition of optimal failure prediction environments.

References

Agrawala, A.K., J.M. Mohr, and R.M. Bryant. 1976. "An Approach to the Workload Characterization Problem." *Computer* 9 (6): 18–32. doi:10.1109/C-M.1976.218610.

Aidemark, J., J. Vinter, P. Folkesson, and J. Karlsson. 2001. "Goofi: Generic Object-Oriented Fault Injection Tool." In \iProceedings of the International Conference on Dependable Systems and Networks, 0083.

Alonso, J., J. Torres, J.L. Berral, and R. Gavalda. 2010. "Adaptive on-Line Software Aging Prediction Based on Machine Learning." In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, 507–16.

Alonso, J., J. Torres, and R. Gavaldà. 2009. "Predicting Web Server Crashes: A Case Study in Comparing Prediction Algorithms." In *Proceedings of the 5th International Conference on Autonomic and Autonomous Systems*, 264–69.

Andrzejak, A., M.M. Moser, and L. Silva. 2007. "Managing Performance of Aging Applications via Synchronized Replica Rejuvenation." In *Proceedings of the Distributed Systems: Operations and Management 18th IFIP/IEEE International Conference on Managing Virtualization of Networks and Services*, 98–109.

Antunes, Nuno, and Marco Vieira. 2010. "Benchmarking Vulnerability Detection Tools for Web Services." In *Proceedings of the IEEE International Conference on Web Services*, 203–10. IEEE. doi:10.1109/ICWS.2010.76.

Arlat, J., M. Aguera, L. Amat, Y. Crouzet, J.C. Fabre, J.C. Laprie, E. Martins, and D. Powell. 1990. "Fault Injection for Dependability Validation: A Methodology and Some Applications." *IEEE Transactions on Software Engineering* 16 (2): 166–82.

Arlat, J., Y. Crouzet, and J.C. Laprie. 1989. "Fault Injection for Dependability Validation of Fault-Tolerant Computing Systems." In *Proceedings of 19th International Symposium on Fault-Tolerant Computing*, 348–55.

Arlat, J., J.-C. Fabre, and M. Rodriguez. 2002. "Dependability of COTS Microkernel-Based Systems." *IEEE Transactions on Computers* 51 (2): 138–63. doi:10.1109/12.980005.

Avizienis, A., J.-C. Laprie, B. Randell, and C. Landwehr. 2004. "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Transactions on Dependable and Secure Computing* 1 (1): 11–33. doi:10.1109/TDSC.2004.2.

Bache, K., and M. Lichman. 2013. *UCI Machine Learning Repository*. University of California, Irvine, School of Information and Computer Sciences. http://archive.ics.uci.edu/ml.

Ball, Thomas, and James R. Larus. 1994. "Optimally Profiling and Tracing Programs." *ACM Transactions in Programming Languages and Systems* 16 (4): 1319–60. doi:10.1145/183432.183527.

Bao, Y., X. Sun, and K.S. Trivedi. 2005. "A Workload-Based Analysis of Software Aging, and Rejuvenation." *IEEE Transactions on Reliability* 54 (3): 541–48.

Barbosa, R., N. Silva, J. Duraes, and H. Madeira. 2007. "Verification and Validation of (real Time) COTS Products Using Fault Injection Techniques." In *Proceedings of the Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*.

Basso, T., R. Moraes, B.P. Sanches, and M. Jino. 2009. "An Investigation of Java Faults Operators Derived from a Field Data Study on Java Software Faults." In *Workshop de Testes E Tolerância a Falhas*, 1–13.

Baum, Eric B., and David Haussler. 1989. "What Size Net Gives Valid Generalization?" *Neural Computation* 1 (1): 151–60.

Bessani, Alysson, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2013. "DepSky: Dependable and Secure Storage in a Cloud-of-Clouds." *IEEE Transactions on Storage Systems* 9 (4): 12:1–12:33. doi:10.1145/2535929.

Bishop, Christopher M., and others. 2006. *Pattern Recognition and Machine Learning*. Vol. 4. 4. Springer New York.

Blum, Avrim L., and Ronald L. Rivest. 1992. "Training a 3-Node Neural Network Is NP-Complete." *Neural Networks* 5 (1): 117–27.

Blumer, Anselm, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. 1987. "Occam's Razor." *Information Processing Letters* 24 (6): 377–80.

Bodik, P., G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, et al. 2005. "Combining Visualization and Statistical Analysis to Improve Operator Confidence and Efficiency for Failure Detection and Localization." In *Proceedings of the Second International Conference on Autonomic Computing*, 89–100.

Bondavalli, A., and L. Simoncini. 1990. "Failure Classification with Respect to Detection." In *Proceedings of the Second IEEE Workshop on Future Trends of Distributed Computing Systems,* 47–53. IEEE Comput. Soc. Press. doi:10.1109/FTDCS.1990.138293.

Bovenzi, A., D. Cotroneo, R. Pietrantuono, and S. Russo. 2011. "Workload Characterization for Software Aging Analysis." In *Proceedings of the IEEE 22nd International Symposium on Software Reliability Engineering,* 240–49. doi:10.1109/ISSRE.2011.18.

Bovenzi, Antonio, Marcello Cinque, Domenico Cotroneo, Roberto Natella, and Gabriella Carrozza. 2011. "OS-Level Hang Detection in Complex Software Systems." *International Journal of Critical Computer-Based Systems* 2 (3/4): 352–77.

Box, George E. P, J. Stuart Hunter, and William Gordon Hunter. 2005. *Statistics for Experimenters : Design, Innovation, and Discovery*. Hoboken, N.J.: Wiley-Interscience.

Boyd, T., and P. Dasgupta. 2002. "Process Migration: A Generalized Approach Using a Virtualizing Operating System." In *22nd International Conference on Distributed Computing Systems, 2002. Proceedings,* 385–92. doi:10.1109/ICDCS.2002.1022276.

Bradley, Andrew P. 1997. "The Use of the Area under the ROC Curve in the Evaluation of Machine Learning Algorithms." *Pattern Recognition* 30: 1145–59.

Calzarossa, M., M. Italiani, and G. Serazzi. 1986. "A Workload Model Representative of Static and Dynamic Characteristics." *Acta Informatica* 23 (3): 255–66. doi:10.1007/BF00289113.

Calzarossa, M., and G. Serazzi. 1993. "Workload Characterization: A Survey." *Proceedings of the IEEE* 81 (8): 1136–50. doi:10.1109/5.236191.

Carreira, J., H. Madeira, and J.G. Silva. 1998. "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers." *IEEE Transactions on Software Engineering* 24 (2): 125–36. doi:10.1109/32.666826.

Carreira, J., H. Madeira, J.G. Silva, and others. 1998. "Xception: Software Fault Injection and Monitoring in Processor Functional Units." *Dependable Computing and Fault Tolerant Systems* 10: 245–66.

Chang, Chih-Chung, and Chih-Jen Lin. 2011. "LIBSVM: A Library for Support Vector Machines." *ACM Trans. Intell. Syst. Technol.* 2 (3): 27:1–27:27. doi:10.1145/1961189.1961199.

Chawla, Nitesh V. 2010. “Data Mining for Imbalanced Datasets: An Overview.” In *Data Mining and Knowledge Discovery Handbook*, edited by Oded Maimon and Lior Rokach, 875–86. Springer US. http://www.springerlink.com/content/v52614081328325x/abstract/.

Chawla, Nitesh V., Nathalie Japkowicz, and Aleksander Kotcz. 2004. “Editorial: Special Issue on Learning from Imbalanced Data Sets.” *ACM SIGKDD Explorations Newsletter* 6 (1): 1. doi:10.1145/1007730.1007733.

Chen, Xin, Charng-Da Lu, and Karthik Pattabiraman. 2014. “Failure Prediction of Jobs in Compute Clouds: A Google Cluster Case Study.” Accessed October 22. http://blogs.ubc.ca/karthik/files/2014/09/rsda14.pdf.

Chillarege, R. 1995. “Orthogonal Defect Classification.” *Handbook of Software Reliability Engineering*.

Chillarege, Ram, Wei-Lun Kao, and Richard G. Condit. 1991. “Defect Type and Its Impact on the Growth Curve.” In *Proceedings of the 13th International Conference on Software Engineering*, 246–55. ICSE ’91. Los Alamitos, CA, USA: IEEE Computer Society Press. http://dl.acm.org/citation.cfm?id=256664.256773.

Chiueh, Susanta Nanda Tzi-cker, and Stony Brook. 2005. “A Survey on Virtualization Technologies.” *RPE Report*, 1–42.

Christmansso, M.H.J., and M. Rimén. 1998. “An Experimental Comparison of Fault and Error Injection.” In *Issre*, 369.

Christmansson, J., and R. Chillarege. 1996. “Generation of an Error Set That Emulates Software Faults Based on Field Data.” In *Annual Symposium on Fault Tolerant Computing*, 304–13.

Citrix. “Citrix XenServer - Efficient Server Virtualization Software.” *Citrix.com*. http://www.citrix.com/.

Committee on National Security Systems. 2010. “National Information Assurance Glossary.” http://www.ncsc.gov/publications/policy/docs/CNSSI_4009.pdf.

Cook, Stephen A. 1971. “The Complexity of Theorem-Proving Procedures.” In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 151–58. ACM. http://dl.acm.org/citation.cfm?id=805047.

Cortes, Corinna, and Vladimir Vapnik. 1995. “Support-Vector Networks.” *Machine Learning* 20 (3): 273–97. doi:10.1007/BF00994018.

“COSBI OpenSourceMark.” http://sourceforge.net/projects/opensourcemark/.

Cotroneo, Domenico, Francesco Fucci, and Roberto Natella. 2012. "Towards a State Driven Workload Generation Framework for Dependability Assessment." In , 25–30. http://www.thinkmind.org/index.php?view=article&articleid=depend_2012_2_10_60023.

Davis, Jesse, and Mark Goadrich. 2006. "The Relationship between Precision-Recall and ROC Curves." In *Proceedings of the 23rd International Conference on Machine Learning*, 233–40. ICML '06. New York, NY, USA: ACM. doi:http://doi.acm.org/10.1145/1143844.1143874.

Dietterich, Thomas. 2002. "Machine Learning for Sequential Data: A Review." In *Structural, Syntactic, and Statistical Pattern Recognition*, edited by Terry Caelli, Adnan Amin, Robert Duin, Dick de Ridder, and Mohamed Kamel, 2396:227–46. Lecture Notes in Computer Science. Springer Berlin / Heidelberg. http://www.springerlink.com/content/av8l8hjl6yc2ya3m/abstract/.

Domeniconi, Carlotta, Chang-Shing Perng, Ricardo Vilalta, and Sheng Ma. 2002. "A Classification Approach for Prediction of Target Events in Temporal Sequences." In *Principles of Data Mining and Knowledge Discovery*, 125–37. Springer. http://link.springer.com/chapter/10.1007/3-540-45681-3_11.

Duraes, J.A., and H.S. Madeira. 2006. "Emulation of Software Faults: A Field Data Study and a Practical Approach." *IEEE Transactions on Software Engineering*, 849–67.

Duraes, J., and H. Madeira. 2003. "Multidimensional Characterization of the Impact of Faulty Drivers on the Operating Systems Behavior." *IEICE Transactions on Information and Systems* 86 (12): 2563–70.

Durães, João, Marco Vieira, and Henrique Madeira. 2004. "Dependability Benchmarking of Web-Servers." In *Computer Safety, Reliability, and Security*, edited by Maritta Heisel, Peter Liggesmeyer, and Stefan Wittmann, 297–310. Lecture Notes in Computer Science 3219. Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/978-3-540-30138-7_25.

Duraes, J., M. Vieira, and H. Madeira. 2004. "Dependability Benchmarking of Web-Servers." *Computer Safety, Reliability, and Security*, 297–310.

Eeckhout, Lieven, Rashmi Sundareswara, Joshua J. Yi, David J. Lilja, and Paul Schrater. 2005. "Accurate Statistical Approaches for Generating Representative Workload Compositions." In *In Proceedings of the 2005 IEEE International Symposium on Workload Characterization*, 55–66.

e-Testing Labs. 2001. *Microsoft: Windows XP Reliability Study*. http://tech-insider.org/windows/research/acrobat/0110.pdf.

Fawcett, Tom. 2006. "An Introduction to ROC Analysis." *Pattern Recognition Letters* 27 (8): 861–74. doi:10.1016/j.patrec.2005.10.010.

Ferrari, D. 1972. "Workload Charaterization and Selection in Computer Performance Measurement." *Computer* 5 (4): 18–24. doi:10.1109/C-M.1972.216939.

Ferrari, Domenico. 1984. "On the Foundations of Artificial Workload Design." In *Proceedings of the 1984 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 8–14. SIGMETRICS '84. New York, NY, USA: ACM. doi:10.1145/800264.809309.

Fonseca, J., M. Vieira, and H. Madeira. 2007. "Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks." In *13th Pacific Rim International Symposium on Dependable Computing, 2007. PRDC 2007*, 365–72. doi:10.1109/PRDC.2007.55.

Frappier, Jonathan. 2014. *VMware vSphere Resource Management Essentials*. Packt Publishing Ltd. https://books.google.it/books?hl=pt-PT&lr=&id=lOHhAgAAQBAJ&oi=fnd&pg=PT8&dq=vmware+vsphere&ots=y50ZJ3CAJE&sig=O-84g8as4EfSM2tTC5X4_SBA_q0.

Fu, Song. 2009. "Failure-Aware Construction and Reconfiguration of Distributed Virtual Machines for High Availability Computing." In , 372–79. IEEE. doi:10.1109/CCGRID.2009.21.

Fu, Song, and Cheng-Zhong Xu. 2007. "Quantifying Temporal and Spatial Fault Event Correlation for Proactive Failure Management." In *IEEE Proceedings of Symposium on Reliable and Distributed Systems (SRDS 07)*.

Garg, S., A. Van Moorsel, K. Vaidyanathan, and K. S. Trivedi. 1998. "A Methodology for Detection and Estimation of Software Aging." In *ISSRE '98: Proceedings of the The Ninth International Symposium on Software Reliability Engineering*, 283. Washington, DC, USA: IEEE Computer Society.

Geman, Stuart, Elie Bienenstock, and René Doursat. 1992. "Neural Networks and the Bias/variance Dilemma." *Neural Comput.* 4 (1): 1–58. doi:10.1162/neco.1992.4.1.1.

Gernert, Dieter. 2009. "Ockham's Razor and Its Improper Use." *Cognitive Systems* 7 (2): 133–38.

Gray, Jim. 1986. "Why Do Computers Stop and What Can Be Done about It?" In *Symposium on Reliability in Distributed Software and Database Systems*, 3–12. Los Angeles, CA, USA. http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf.

Gray, Jim. 1993. *The Benchmark Handbook : For Database and Transaction Processing Systems*. San Mateo, Calif.: M. Kaufmann Publishers.

Gross, Kenny C., Vatsal Bhardwaj, and Randy Bickford. 2002. "Proactive Detection of Software Aging Mechanisms in Performance Critical Computers." *Software Engineering Workshop, Annual IEEE/NASA Goddard*, 17. doi:http://doi.ieeecomputersociety.org/10.1109/SEW.2002.1199445.

Grottke, Michael, and Kishor S. Trivedi. 2005. "A Classification of Software Faults." *Journal of Reliability Engineering Association of Japan* 27 (7): 425–38.

Gunawi, Haryadi S., Thanh Do, Joseph M. Hellerstein, Ion Stoica, Dhruba Borthakur, and Jesse Robbins. 2011. "Failure as a Service (faas): A Cloud Service for Large-Scale, Online Failure Drills." *University of California, Berkeley, Berkeley* 3. http://digitalassets.lib.berkeley.edu/techreports/ucb/text/EECS-2011-87.pdf.

Gunneflo, U., J. Karlsson, and J. Torin. 1989. "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-Ion Radiation." In *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*, 340–47.

Gurumurthy, Kaushik, Guruswamy Namasivayam, Sunil Kutty, Michael G. Tricker, and Angel Sarmento Calvo. 2015. "Cloud Deployment Infrastructure Validation Engine." http://www.freepatentsonline.com/y2015/0052402.html.

Guyon, Isabelle, and André Elisseeff. 2003. "An Introduction to Variable and Feature Selection." *The Journal of Machine Learning Research* 3: 1157–82.

Hand, David J. 2012. "Assessing the Performance of Classification Methods." *International Statistical Review*, no – no. doi:10.1111/j.1751-5823.2012.00183.x.

Hatonen, Kimmo, Mika Klemettinen, Heikki Mannila, Pirjo Ronkainen, and Hannu Toivonen. 1996. "TASA: Telecommunication Alarm Sequence Analyzer or How to Enjoy Faults in Your Network." In *Network Operations and Management Symposium, 1996., IEEE*, 2:520–29. IEEE. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=539622.

Hochreiter, Sepp, and Klaus Obermayer. 2006. "Nonlinear Feature Selection with the Potential Support Vector Machine." In *Feature Extraction*, edited by Isabelle Guyon, Masoud Nikravesh, Steve Gunn, and Lotfi Zadeh, 207:419–38. Studies in Fuzziness and Soft Computing. Springer Berlin / Heidelberg. http://www.springerlink.com/content/h8375v6464111507/abstract/.

Hoffmann, G.A. 2006. "Failure Prediction in Complex Computer Systems: A Probabilistic Approach." Shaker.

Hoffmann, GA, and M. Malek. 2006. "Call Availability Prediction in a Telecommunication System: A Data Driven Empirical Approach." In *25th Symposium on Reliable Distributed Systems*.

Hoffmann, G.A., F. Salfner, and M. Malek. 2004. *Advanced Failure Prediction in Complex Software Systems*. Citeseer.

Hoffmann, G.A., K.S. Trivedi, and M. Malek. 2006. "A Best Practice Guide to Resource Forecasting for the Apache Webserver." In *IEEE Proceedings of the 12th International Symposium Pacific Rim Dependable Computing (PRDC'06). University of California, Riverside, USA*.

Hoffmann, G.A., K.S. Trivedi, and M. Malek. 2007. "A Best Practice Guide to Resource Forecasting for Computing Systems." *Reliability, IEEE Transactions on* 56 (4): 615–28. doi:10.1109/TR.2007.909764.

Hoffmann, Günther A. 2004. "Adaptive Transfer Functions in Radial Basis Function (RBF) Networks." In *Computational Science - ICCS 2004*, edited by Marian Bubak, Geert Dick Albada, Peter M. A. Sloot, and Jack Dongarra, 3037:682–86. Berlin, Heidelberg: Springer Berlin Heidelberg. http://www.springerlink.com/index/10.1007/978-3-540-24687-9_102.

Hsueh, M.C., T.K. Tsai, and R.K. Iyer. 1997. "Fault Injection Techniques and Tools." *Computer* 30 (4): 75–82.

Hudak, JJ, B.H. Suh, DP Siewiorek, and Z. Segall. 1993. "Evaluation and Comparison of Fault-Tolerant Software Techniques." *Reliability, IEEE Transactions on* 42 (2): 190–204.

Hughes, G. 1968. "On the Mean Accuracy of Statistical Pattern Recognizers." *IEEE Transactions on Information Theory* 14 (1): 55–63. doi:10.1109/TIT.1968.1054102.

Hughes, G.F., J.F. Murray, K. Kreutz-Delgado, and C. Elkan. 2002. "Improved Disk-Drive Failure Warnings." *Reliability, IEEE Transactions on* 51 (3): 350–57. doi:10.1109/TR.2002.802886.

Irrera, I. 2013. "Virtualization Impact Assessment: Complete Set of Results." http://eden.dei.uc.pt/~ivano.

Irrera, I., J. Duraes, and M. Vieira. 2014. "On the Need for Training Failure Prediction Algorithms in Evolving Software Systems." In *To Appear in 15th IEEE International Symposium on High Assurance Systems Engineering (HASE'14)*. Miami, Florida, USA.

Irrera, I., J. Durães, M. Vieira, and H. Madeira. 2010. "Towards Identifying the Best Variables for Failure Prediction Using Injection of Realistic Software Faults." In *Pacific Rim International Symposium on Dependable Computing (PRDC 2010)*, 3–10.

Irrera, Ivano, Joao Duraes, Henrique Madeira, and Marco Vieira. 2013. "Assessing the Impact of Virtualization on the Generation of Failure Prediction Data." In *Latin-American Symposium on Dependable Computing*, 0:92–97. Los Alamitos, CA, USA: IEEE Computer Society. doi:10.1109/LADC.2013.24.

Irrera, Ivano, Carlos Pereira, and Marco Vieira. 2013. "The Time Dimension in Predicting Failures: A Case Study." In *Latin-American Symposium on Dependable Computing*, 0:86–91. Los Alamitos, CA, USA: IEEE Computer Society. doi:10.1109/LADC.2013.25.

Irrera, Ivano, Marco Vieira, and Joao Duraes. 2015. "Adaptive Failure Prediction for Computer Systems: A Framework and a Case Study." In *IEEE 16th International Symposium on High-Assurance Systems Engineering*, 142–49. doi:10.1109/HASE.2015.29.

Irrera, I., and M. Vieira. 2014. "A Practical Approach for Generating Failure Data for Assessing and Comparing Failure Prediction Algorithms." In *To Appear in PRDC'14 Proceedings*. Singapore.

Jain, Jeenia, and Ramandeep Singh. 2014. "Improving Service Reliability in Cloud Computing Environment." Accessed October 22. http://www.ijser.org/researchpaper%5CImproving-Service-Reliability-in-Cloud-Computing-Environment.pdf.

Jeni, László, Jeffrey F. Cohn, Fernando De La Torre, and others. 2013. "Facing Imbalanced Data–Recommendations for the Use of Performance Metrics." In *Affective Computing and Intelligent Interaction (ACII), 2013 Humaine Association Conference on*, 245–51. IEEE. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6681438.

John, George H., Ron Kohavi, Karl Pfleger, and others. 1994. "Irrelevant Features and the Subset Selection Problem." In *Machine Learning: Proceedings of the Eleventh International Conference*, 121–29. https://www.google.com/books?hl=it&lr=&id=cEqjBQAAQBAJ&oi=fnd&pg=PA121&dq=Irrelevant+features+and+the+subset+selection+problem&ots=E1ounfz4CN&sig=XiZ1_OYGyrmRHOjiLJt6zJ40P3g.

Kalakech, Ali, Karama Kanoun, Yves Crouzet, and Jean Arlat. 2015. "Benchmarking The Dependability of Windows NT4, 2000 and XP." Accessed May 27. https://homepages.laas.fr/~arlat/documents/03501/03501.pdf.

Kalyanakrishnam, M., Z. Kalbarczyk, and R. Iyer. 1999. "Failure Data Analysis of a LAN of Windows NT Based Computers." In *18th Symposium on Reliable Distributed Systems*, 178–87. doi:10.1109/RELDIS.1999.805094.

Kanawati, G.A., N.A. Kanawati, and J.A. Abraham. 1992. "FERRARI: A Tool for the Validation of System Dependability Properties." In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, 336–44.

Karlsson, J., P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger. 1998. "Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture." *DEPENDABLE COMPUTING AND FAULT TOLERANT SYSTEMS* 10: 267–88.

Kennedy, James. 2010. "Particle Swarm Optimization." In *Encyclopedia of Machine Learning*, 760–66. Springer. http://link.springer.com/10.1007/978-0-387-30164-8_630.

Khanna, G., K. Beaty, G. Kar, and A. Kochut. 2006. "Application Performance Management in Virtualized Server Environments." In , 373–81. IEEE. doi:10.1109/NOMS.2006.1687567.

Ko, Andrew J., Bryan Dosono, and Neeraja Duriseti. 2014. "Thirty Years of Software Problems in the News." In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, 32–39. ACM. http://dl.acm.org/citation.cfm?id=2593719.

Koopman, P., and H. Madeira. 1999. "Dependability Benchmarking & Prediction: A Grand Challenge Technology Problem." *Real-Time Mission-Critical Systems: Grand Challenge Problems, Phoenix, Arizona USA*, November.

Koopman, P., J. Sung, C. Dingman, D. Siewiorek, and T. Marz. 1997. "Comparing Operating Systems Using Robustness Benchmarks." In *SRDS*, 72.

Laprie, Jean-Claude. 2005. "Resilience for the Scalability of Dependability." In *Network Computing and Applications, Fourth IEEE International Symposium on*, 5–6. IEEE. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1565929.

Lee, I., and R.K. Iyer. 1995. "Software Dependability in the Tandem GUARDIAN System." *IEEE Transactions on Software Engineering* 21 (5): 455–67.

Levy, D., and R. Chillarege. 2003. "Early Warning of Failures through Alarm Analysis a Case Study in Telecom Voice Mail Systems." In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, 271–80.

Liang, Y., Y. Zhang, M. Jette, Anand Sivasubramaniam, and R. Sahoo. 2006. "BlueGene/L Failure Analysis and Prediction Models." In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, 425–34. doi:10.1109/DSN.2006.18.

Li, Lei, Kalyanaraman Vaidyanathan, and Kishor S. Trivedi. 2002. "An Approach for Estimation of Software Aging in a Web Server." In *ISESE*, 91–102. http://doi.ieeecomputersociety.org/10.1109/ISESE.2002.1166929.

Lin, T.T.Y., and D.P. Siewiorek. 1990. "Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis." *Reliability, IEEE Transactions on* 39 (4): 419–32.

Liu, Huan, and Rudy Setiono. 1997. "Feature Selection and Classification-a Probabilistic Wrapper Approach." In *Proceedings of 9th International Conference on Industrial and Engineering Applications of AI and ES*, 419–24. https://www.google.com/books?hl=it&lr=&id=0xdubWCSbv0C&oi=fnd&pg=PA419&dq=Feature+Selection+And+Classification+A+Probabilistic+Wrapper+Approach&ots=nzsPEJYRqh&sig=BgBpB5cQTeE_7WNdCBriu6cxwWc.

Liu, H., and L. Yu. 2005. "Toward Integrating Feature Selection Algorithms for Classification and Clustering." *Knowledge and Data Engineering, IEEE Transactions on* 17 (4): 491–502.

Luke Stackwalker. http://lukestackwalker.sourceforge.net/.

Lyu, Michael R., and others. 1996. *Handbook of Software Reliability Engineering*. Vol. 222. IEEE computer society press CA. http://trustworthy.googlecode.com/svn/trunk/doc/Handbook_of_Software_Reliability_Engineering.pdf.

Machida, F., M. Kawato, and Y. Maeno. 2010. "Redundant Virtual Machine Placement for Fault-Tolerant Consolidated Server Clusters." In *2010 IEEE Network Operations and Management Symposium (NOMS)*, 32–39. doi:10.1109/NOMS.2010.5488431.

Madeira, H., D. Costa, and M. Vieira. 2000. "On the Emulation of Software Faults by Software Fault Injection." In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, 417–26.

Magalhaes, J.P., and L. Moura Silva. 2013. "A Framework for Self-Healing and Self-Adaptation of Cloud-Hosted Web-Based Applications." In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom)*, 1:555–64. doi:10.1109/CloudCom.2013.80.

Magalhaes, J.P., and L.M. Silva. 2012. "Anomaly Detection Techniques for Web-Based Applications: An Experimental Study." In *2012 11th IEEE International Symposium on Network Computing and Applications (NCA)*, 181–90. doi:10.1109/NCA.2012.27.

Martinez, M., D. de Andres, J.-C. Ruiz, and J. Friginal. 2014. "From Measures to Conclusions Using Analytic Hierarchy Process in Dependability Benchmarking." *IEEE Transactions on Instrumentation and Measurement* 63 (11): 2548–56. doi:10.1109/TIM.2014.2348632.

Matias, R., K.S. Trivedi, and P.R.M. Maciel. 2010. "Using Accelerated Life Tests to Estimate Time to Software Aging Failure." In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, 211–19.

Maxion, R.A., and K.M.C. Tan. 2000. "Benchmarking Anomaly-Based Detection Systems." In *Proceedings International Conference on Dependable Systems and Networks, 2000. DSN 2000*, 623–30. doi:10.1109/ICDSN.2000.857599.

Mendes, N., J. Duraes, and H. Madeira. 2011. "Benchmarking the Security of Web Serving Systems Based on Known Vulnerabilities." In *2011 5th Latin-American Symposium on Dependable Computing (LADC)*, 55–64. doi:10.1109/LADC.2011.14.

Mendes, N., H. Madeira, and J. Duraes. 2014. "Security Benchmarks for Web Serving Systems." In *2014 IEEE 25th International Symposium on Software Reliability Engineering (ISSRE)*, 1–12. doi:10.1109/ISSRE.2014.38.

Monperrus, Martin. 2014. "Principles of Antifragile Software." *arXiv:1404.3056 [cs]*, April. http://arxiv.org/abs/1404.3056.

Moraes, R., J. Duraes, R. Barbosa, E. Martins, and H. Madeira. 2007. "Experimental Risk Assessment and Comparison Using Software Fault Injection."

Moro, A., E. Mumolo, and M. Nolich. 2009. "Ergodic Continuous Hidden Markov Models for Workload Characterization." In *Proceedings of 6th International Symposium on Image and Signal Processing and Analysis, 2009. ISPA 2009*, 99–104.

Murray, J.F., G.F. Hughes, and K. Kreutz-Delgado. 2003. "Hard Drive Failure Prediction Using Non-Parametric Statistical Methods." In *Proceedings of ICANN/ICONIP*.

Nagarajan, Arun Babu, Frank Mueller, Christian Engelmann, and Stephen L. Scott. 2007. "Proactive Fault Tolerance for HPC with Xen Virtualization." In , 23. ACM Press. doi:10.1145/1274971.1274978.

NASA. "NASA-STD 8739.8 - Standard for Software Assurance." http://www.hq.nasa.gov.

Nassar, Fares A., and Dorothy M. Andrews. 1985. *A Methodology for Analysis of Failure Prediction Data*. Center for Reliable Computing, Computer Systems Laboratory, Depts. of Electrical Engineering and Computer Science, Stanford University.

Natella, R., and D. Cotroneo. 2010. "Emulation of Transient Software Faults for Dependability Assessment: A Case Study." In *2010 European Dependable Computing Conference*, 23–32.

Natella, R., D. Cotroneo, J. Duraes, and H. Madeira. 2010. "Representativeness Analysis of Injected Software Faults in Complex Software." In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 437–46.

Natella, Roberto, Domenico Cotroneo, Joao A. Duraes, and Henrique S. Madeira. 2013. "On Fault Representativeness of Software Fault Injection." *IEEE Transactions on Software Engineering* 39 (1): 80–96. doi:10.1109/TSE.2011.124.

Neto, A.A., and M. Vieira. 2011. "Trustworthiness Benchmarking of Web Applications Using Static Code Analysis." In *2011 Sixth International Conference on Availability, Reliability and Security (ARES)*, 224–29. doi:10.1109/ARES.2011.37.

OMG System Assurance Task Force. "OMG Software Assurance (SwA) Special Interest Group (SIG)." http://swa.omg.org.

Oppenheimer, David. 2003. “Why Do Internet Services Fail, and What Can Be Done about It?” In *Symposium on Internet Technologies and Systems (USITS’03)*. http://lambda.csail.mit.edu/~chet/papers/others/r/recovery/ms-final-single.pdf.

Otsuka, Hiroshi, Kaustubh Joshi, Matti Hiltunen, Scott Daniels, and Yasuhide MATSUMOTO. 2014. “Online Failure Prediction with Accurate Failure Localization in Cloud Infrastructures.” *IEICE Technical Report. SC, Services Computing* 113 (496): 7–12.

Pitakrat, T., A. Van Hoorn, and L. Grunske. 2014. “Increasing Dependability of Component-Based Software Systems by Online Failure Prediction (Short Paper).” In *Dependable Computing Conference (EDCC), 2014 Tenth European*, 66–69. doi:10.1109/EDCC.2014.28.

Polze, A., P. Troger, and F. Salfner. 2011. “Timely Virtual Machine Migration for Pro-Active Fault Tolerance.” In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on*, 234–43.

RAR Lab. “WinRAR Archiver.” http://www.rarlab.com/.

Reiser, Hans P., and Rudiger Kapitza. 2007. “Hypervisor-Based Efficient Proactive Recovery.” In , 83–92. IEEE. doi:10.1109/SRDS.2007.25.

Rimal, Bhaskar Prasad, Eunmi Choi, and Ian Lumb. 2009. “A Taxonomy and Survey of Cloud Computing Systems.” In *INC, IMS and IDC, 2009. NCM’09. Fifth International Joint Conference on*, 44–51. Ieee. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5331755.

Rose, Robert. 2004. *Survey of System Virtualization Techniques*.

SAFECode - Software Assurance Forum for Excellence in Code. 2008. “Software Assurance: An Overview of Current Industry Best Practices.” http://www.safecode.org.

Sahoo, J., S. Mohapatra, and R. Lath. 2010. “Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues.” In *2nd Int’l Conference on Computer and Network Technology*, 222–26. doi:10.1109/ICCNT.2010.49.

Salfner, F. 2006. *Modeling Event-Driven Time Series with Generalized Hidden Semi-Markov Models*. Professoren des Inst. f\ür Informatik.

Salfner, Felix, Maren Lenk, and Miroslaw Malek. 2010. “A Survey of Online Failure Prediction Methods.” *ACM Comput. Surv.* 42 (3): 10:1–10:42. doi:10.1145/1670679.1670680.

Salfner, F., G. Hoffmann, and M. Malek. 2005. “Prediction-Based Software Availability Enhancement.” *Self-Star Properties in Complex Information Systems*, 143–57.

Salfner, F., M. Lenk, and M. Malek. 2010. "A Survey of Online Failure Prediction Methods." *ACM Computing Surveys (CSUR)* 42 (3): 1–42.

Salfner, F., and M. Malek. 2005. "Proactive Fault Handling for System Availability Enhancement."

Salfner, F., and M. Malek. 2007. "Using Hidden Semi-Markov Models for Effective Online Failure Prediction." *26th Int'l Symposium on Reliable Distributed Systems (SRDS 2007).*

Salfner, F., and M. Malek. 2010. "Architecting Dependable Systems with Proactive Fault Management." *Architecting Dependable Systems VII*, 171–200.

Sanches, B.P., T. Basso, and R. Moraes. 2011. "J-SWFIT: A Java Software Fault Injection Tool." In *2011 Latin-American Symposium on Dependable Computing*, 106–15.

Sauer, Chris. 1993. *Why Information Systems Fail: A Case Study Approach*. Oxfordshire, UK, UK: Alfred Waller Ltd., Publishers.

Schölkopf, Bernhard, and Alexander J. Smola. 2002. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and beyond*. MIT press. https://books.google.com/books?hl=it&lr=&id=y8ORL3DWt4sC&oi=fnd&pg=PR13&dq=optimization+machine+learning+parameters&ots=bKvScwN8Iz&sig=08nyR6czgbQyfC2AUvLcf0F_Bkg.

Siddaway, Richard. 2012. *Powershell and WMI*. Manning. http://www.manning.com.p.hostingprod.com/siddaway2/PSaWMIchapter1sample.pdf.

Simoncini, L. 2009. "Resilient Computing: An Engineering Discipline." In *IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009*, 1–1. doi:10.1109/IPDPS.2009.5160867.

Smith, Wayne D. 2000. *TPC-W: Benchmarking an Ecommerce Solution*.

Sonoda, Masataka, Shinji Kikuchi, Yukihiro Watanabe, Hiroshi Otsuka, and Yasuhide Matsumoto. 2012. "Online Failure Prediction in Cloud Datacenters by Real-Time Message Pattern Learning." In *Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, 504–11. CLOUDCOM '12. Washington, DC, USA: IEEE Computer Society. doi:10.1109/CloudCom.2012.6427566.

"Standard Performance Evaluation Corporation (SPEC)." Www.spec.org.

Sullivan, M., and R. Chillarege. 1991. "Software Defects and Their Impact on System Availability-a Study of Field Failures in Operating Systems." In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, 2–9.

"Support - Windows Help." 2015. *Windows.microsoft.com*. Accessed May 9. http://windows.microsoft.com/en-us/windows/support.

Taleb, Nassim Nicholas. 2012. *Antifragile: Things That Gain from Disorder*. Random House Incorporated.

Tanenbaum, Andrew, and Maarten Van Steen. 2007. *Distributed Systems*. Pearson Prentice Hall. http://www.cs.helsinki.fi/u/alanko/hj/K06/kalvokopiot/ch1_p6.pdf.

"Tech Insider - Various Studies." http://tech-insider.org/windows/.

"Tomcat Version 7 - Changelog." https://tomcat.apache.org/tomcat-7.0-doc/changelog.html.

"Transaction Processing Performance Council (TPC)." Www.tpc.org.

Trivedi, K.S., Dong Seong Kim, and R. Ghosh. 2009. "Resilience in Computer Systems and Networks." In *IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009*, 74–77.

Tsai, Timothy K., and Ravishankar K. Iyer. 1995. "Ftape: A Fault Injection Tool to Measure Fault Tolerance." http://ntrs.nasa.gov/search.jsp?R=19950059069.

Usenix, and Carnegie Mellon University (CMU). 2006. "Computer Failure Data Repository." https://www.usenix.org/cfdr.

Vaidyanathan, K., R.E. Harper, S.W. Hunter, and K.S. Trivedi. 2001. "Analysis and Implementation of Software Rejuvenation in Cluster Systems." In *ACM SIGMETRICS Performance Evaluation Review*, 29:62–71.

Vaidyanathan, K., and K.S. Trivedi. 1999. "A Measurement-Based Model for Estimation of Resource Exhaustion in Operational Software Systems." In *10th Int'l Symposium on Software Reliability Engineering (ISSRE 1999)*, 84–93.

Vaidyanathan, K., and K.S. Trivedi. 2001. "Extended Classification of Software Faults Based on Aging." In *Proceedings of the Twelfth International Symposium on Software Reliability Engineering (ISSRE)*, 27–28.

Van Rijsbergen, C. 1979. *Information Retrieval*. 2d ed. London; Boston: Butterworths.

Vieira, Marco, and Henrique Madeira. 2005. "Towards a Security Benchmark for Database Management Systems." In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, 592–601. IEEE. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1467833.

Vieira, M., and H. Madeira. 2003. "A Dependability Benchmark for OLTP Application Environments." In *Proceedings of the 29th International Conference on Very Large Data Bases-Volume 29*, 742–53.

Vieira, M., H. Madeira, I. Irrera, and M. Malek. 2009. "Fault Injection for Failure Prediction Methods Validation." In *40th Int'l Conference on Dependable Systems and Networks (5th Workshop on Hot Topics in System Dependability)*.

Vukotic, Aleksa, and James Goodwill. 2011. *Apache Tomcat 7*. 1st ed. Berkely, CA, USA: Apress.

Wang, John, ed. 2008. *Encyclopedia of Data Warehousing and Mining, Second Edition*. IGI Global. http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/978-1-60566-010-3.

Watanabe, Y., H. Otsuka, M. Sonoda, S. Kikuchi, and Y. Matsumoto. 2012. "Online Failure Prediction in Cloud Datacenters by Real-Time Message Pattern Learning." In *2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, 504–11. doi:10.1109/CloudCom.2012.6427566.

Watanabe, Yukihiro, and Yasuhide Matsumoto. 2014. "Online Failure Prediction in Cloud Datacenters." *FUJITSU Sci. Tech. J* 50 (1): 66–71.

Weiss, G.M. 1999. "Timeweaver: A Genetic Algorithm for Identifying Predictive Patterns in Sequences of Events." In *Proceedings of the Genetic and Evolutionary Computation Conference*, 718–25.

Wolski, R., N. Spring, and C. Peterson. 1997. "Implementing a Performance Forecasting System for Metacomputing: The Network Weather Service." In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing (CDROM)*, 1–19.

Zemouri, Ryad, and Noureddine Zerhouni. 2011. "Autonomous and Adaptive Procedure for Cumulative Failure Prediction." *Neural Computing and Applications* 21 (2): 319–31. doi:10.1007/s00521-011-0585-7.

Zheng, Zjian. 1993. *A Benchmark for Classifier Learning*. University of Sidney.

Zweig, M H, and G. Campbell. 1993. "Receiver-Operating Characteristic (ROC) Plots: A Fundamental Evaluation Tool in Clinical Medicine." *Clinical Chemistry* 39 (4): 561–77.