Chapter 11
Field Studies on Resilience: Measurements and Repositories

Joao Duraes, José Fonseca, Henrique Madeira and Marco Vieira

Abstract This chapter is devoted to field studies and the aspects related to this kind of measurements. The importance of measurements collected from the operational scenarios is discussed, and two case studies are presented. Field measurements are closely tied to data repositories, and this chapter presents an overview of some field data repositories available to the public.

11.1 Introduction

Field measurements refer to observations of systems in the operational phase, i.e., systems that are actually in use. The results obtained from these observations have the very important characteristic of being realistic: the operation conditions and environment, and the workload are not mere experimental approximations. Very often, field studies are not representative as there is no guarantee that all possible, important system configurations have been observed. Nevertheless, field measurements and field data are a unique and very important source of information for researchers when studying resilience properties, such as availability, reliability and robustness.

J. Duraes (✉)
DEI/CISUC, Polytechnic Institute of Coimbra, 3030-290 Coimbra, Portugal
e-mail: jduraes@isec.pt

J. Fonseca
DEI/CISUC, University of Coimbra & UDI, Polytechnic Institute of Guarda,
3030-290 Coimbra, Portugal
e-mail: josefonseca@ipg.pt

H. Madeira · M. Vieira
DEI/CISUC, University of Coimbra, 3030-290 Coimbra, Portugal
e-mail: henrique@dei.uc.pt

M. Vieira
e-mail: mvieira@dei.uc.pt

K. Wolter et al. (eds.), *Resilience Assessment and Evaluation of Computing Systems*,
DOI: 10.1007/978-3-642-29032-9_11,

There are basically two main driving forces behind the collection of field data: development and research. The first is committed to the improvement of specific systems and to solve problems on those specific systems that are discovered during the operational phase. The second driving force aims to understand the issues related to systems reliability and dependability and to propose new techniques to increase the reliability of non-specific (non vendor-specific) systems. A third driving force is a market-driven one, to promote awareness of a given product (e.g., network providers, such as sprint and AT&T, publish their performance and dependability data to promote the company and attract new customers). However, the first two driving forces are those more relevant to research works.

The research driving force, although not tied to specific vendors or industry goals, is necessarily dependent on the existence of data. These data is mainly that which was collected by users or operators and is not related to any research goal. Thus, so far, the main origin of field data is the occurrence of incidents. This fact has an overwhelming impact on the nature of the available data, which is mainly related to computer failures and security incidents. To demonstrate the importance of field measurement and what can be achieved, in this chapter we present two case studies: the first on software faults and the second related to security vulnerabilities.

The most complex and error prone components of computer-based systems are the software. Understanding software faults is essential to devise mechanisms to mitigate faults existing in software. Thus, the first case study presented in this chapter is a field study on software faults aimed at the characterization of software faults for emulation and fault injection purposes.

Security issues are currently one of the major concerns surrounding software systems. Networking is one of the scenarios that most exposes a system to the general public and potential malicious users and attacks, representing a high relation with security-related incidents. Web-based systems are currently the basis of the majority of network-enabled systems. The second case study presented is thus related to security vulnerabilities.

Although field data (field measurements) are highly relevant to the research community to understand and improve computer-based systems robustness, reliability and security, the availability of such data remains hard to guarantee. The few data available are based on open-source projects and published research works. The importance of field data is widely recognized among researchers as shown in workshops such as RAF07: Reliability Analysis of System Failure Data organized by Microsoft Research in Cambridge and Darmstadt University in 2007. Each open-source development team or research team presents its own data and its own view. One important initiative to mitigate the scarcity and fragmented view of field data is the development of public repositories, to store data and results based on that data originating from many sources and teams. We include in this chapter a brief overview of available data repositories.

The outline of the chapter is as follows. Section 11.2 presents a field study on software faults. Section 11.3 presents a field study on security vulnerabilities. An overview of field data repositories is presented in Sect. 11.4. Section 11.5 concludes this chapter.

11.2 Case Study 1: Field Data on Software Faults

This section presents a field study on real software faults. This case study was conducted to understand the nature of faults, and to obtain a classification scheme usable for fault injection. Injecting faults is a time-proved method of validating fault tolerant mechanisms and assess system robustness. Given the relevance of software faults, it is very relevant to be able to inject software faults. The usefulness of fault injection is tied to the representativeness of the faults injected. To that aim, we need to understand what exactly is a software fault (a clear, but detailed description usable for automated fault injection), and obtain information on the types of faults that represent the faults more common in the operational scenario. The case study presented here is a summarized description of that field study. More details can be found in [315]. A technique to emulate software faults at the binary executable was proposed based on the findings of this study (G-SQFIT, see [315]), however, the details of such technique do not fit in a field study description and it is not presented here.

Section 11.2.1 presents the source of the software faults used in this case study and details the methodology used for the classification of the faults. Section 11.2.2 presents a first overview of the fault distribution and makes a comparative analysis with the field study done by Christmansson and Chillarege in [213] using the ODC classification [205, 206] scheme. Section 11.2.3 presents an overview of the details classification of the collected faults. Some conclusions about this field study are presented in Sect. 11.2.4.

11.2.1 Sources of Real Software Faults and Classification Methodology

To address the representativeness issue of our study, we collected a large set of real software faults from software used in the field. The goal was to improve the knowledge about the exact nature of faults and their occurrence distribution using data from the real operational scenario. More specifically, the software faults that are pertinent to emulate by fault injection are those that originated in the coding phase and eluded the testing procedures and go with the deployed product.

The information source used in our work was a set of diff/patch files for several open source programs. The diff/patch files contain source code corrections for faults discovered after the software was released. By manual inspection of those files we were able to extract information to understand and classify software faults. From those diff/patch files, a total of 668 faults were analyzed. Table 11.1 presents a summary of the programs used in this study. It is worth noting that these programs encompass a broad range of program types: both user programs (including interactive and command line programs) and operating system (Linux kernels) were used.

The total number of faults collected for each program is dependent on the program age, maturity and the user community size. Some of the programs (e.g. Bash) are

Table 11.1 Source of the field data

Programs	Source location	Description	# faults
CDEX	http://sourceforge.net/projects/cdexos/	CD Digital audio data extractor	11
Vim	http://www.vim.org	Improved version of the UNIX vi	249
FreeCiv	http://www.freeciv.org	Multiplayer strategy game	53
pdf2h	http://sourceforge.net/projects/pdftohtml/	pdf to html format translator	20
GAIM	http://sourceforge.net/projects/gaim/	All-in-one multi-protocol IM client	23
Joe	http://sourceforge.net/projects/joe-editor/	Text editor similar to Wordstar®	78
ZSNES	http://sourceforge.net/projects/zsnes/	SNES/Super Famicom emulator	3
Bash	http://cnswww.cns.cwru.edu/~chet/bash/bas	GNU Project's Bourne Again	2
LKernel	http://www.kernel.org	Linux kernels 2.0.39 and 2.2.22	93
Firebird	http://sourceforge.net/projects/firebird/	Cross-platform RDBMS engine	2
MingW	http://www.mingw.org/	Minimalist GNU for Windows	60
ScummV	http://sourceforge.net/projects/scummvm	Iterpreter for adventure engines	74
Total faults collected			668

in a mature phase and have few recent faults; other programs (e.g. VIM) are still in the maturation phase and have a large user community that provides many fault reports. The notion of fault requires the notion of correctness. Generally speaking, the software is correct if it conforms to the user needs, as specified in the software requirements. However those might be wrong. For the purpose of this work, it was assumed that the requirements and specification are correct. Thus, a software fault means that the code is not correct somehow (i.e., it does not implement the specification in some particular aspect) because the code does not contain the instructions that should have.

The approach used to analyze and classify the faults was the following:

1. First we classified the faults according to the Orthogonal Defect Classification scheme (ODC) [205, 206]. The use of general and well accepted fault classification is the best way to make our results available for the research community and it allows us to compare our results with previous field studies.
2. In a second step we grouped the faults in each ODC class according to the nature of the defect, defined from a building block programming point of view. That is, for each ODC class a software fault is further characterized by one or more programming language constructs that is either missing, wrong or in excess. Programming language constructs may be statements, expressions, function calls, etc. A fault may then fall in one of three possible types: missing construct, Wrong construct, and Extraneous construct. This is very relevant to fault emulation/injection since emulating an omission (missing construct) is substantially different from emulating a wrong construct (e.g., erroneous expression).
3. In a last step, faults were further described and grouped into specific types. Each type is defined according to the language construct and program context surrounding the fault location. This description refinement is also particularly relevant for fault injection purposes since it helps (a) the identification of suitable

locations in the target code, and (b) the code modifications necessary to emulate a given fault type.

The resulting final classification can be viewed as an extension to ODC and is used to define fault emulation operators (each operator emulates one specific type of faults).

11.2.2 ODC Classification and General Analysis

According to the Orthogonal Defect Classification, a software fault is characterized by the change in the code that is necessary to correct it, i.e., to put the code consistent with the specification, which is assumed to be correct in our case. From the list of ODC types, the following are directly related to the code and relevant to our work:

- Assignment: value(s) assigned incorrectly or not assigned at all.
- Checking: missing or incorrect validation of data and conditional statements, wherever these checks and conditions may appear (e.g., an incorrect loop condition).
- Interface: errors in the interaction among components, modules, device drivers, functions calls, and similar.
- Timing/serialization: missing or incorrect serialization of shared resources.
- Algorithm: incorrect or missing implementation that can be fixed only by (re)implementing an algorithm or data structure without the need of a design change.
- Function: affects a sizeable amount of code and refers to capability that is either implemented incorrectly or not implemented at all.

As field data available to us did not include any information on timing or serialization properties, we did not consider the Timing/serialization ODC type. The mapping of the faults into one of the remaining ODC types was straightforward with the exception of the Function type which required a more detailed analysis of the code in order to figure out whether the correction of the fault has required a design change or not. Due to the decentralized nature of the software development methodology of open source projects, we didn't have direct information on redesign decisions, which forced us to a more detailed analysis of the faults identified as candidates for the Function ODC type. Table 11.2 presents the distribution of faults across the five ODC fault types addressed in this work.

One interesting topic to both the theme of field-based works and to the theme of software faults is the comparison of our results with other available field studies that also used ODC to classify field-discovered faults. We compared our fault distribution with the one presented in [213] as that work is the one most closely related to our own. Because that work included Time/Serialization faults, we removed that particular type from the comparison and normalize all the percentages leaving so that a direct comparison could be made. Table 11.2 presents this comparison (values shown in parenthesis are those from [213] after normalization.

It is relevant that both our data and that presented in [213] show the same trend in the fault distribution across ODC fault types: assignment faults have approximately

Table 11.2 Fault distribution across ODC types

ODC type	# faults	ODC distribution (%)	
Assignment	143	21.4	(21.98)
Checking	167	25.0	(17.48)
Interface	49	7.3	(8.17)
Algorithm	268	40.1	(43.41)
Function	41	6.1	(8.74)

the same weight as Checking faults; Interface and Function faults are clearly the less frequent ones; and Algorithm are the dominant faults. All ODC classes have approximately the same weight in both works. The fact that independent research works obtained a similar fault distribution suggests that this distribution is representative of programs in general and gives us confidence in our results. Also, the programs analyzed in [213] (large database and operating system code) were quite different from the ones used in our study, suggesting that this fault distribution across ODC types is reasonably independent from the nature of the program. Although more field studies should be conducted to consolidate this conclusion, these results suggest that fault injection experiments should take this fault distribution trend into consideration to improve representativeness.

Table 11.3 presents the fault distribution observed for each individual program used in this study. To observe a trend in fault distribution across programs, only those programs with a significant number of faults should be considered (the number of faults is presented in the first row). Nevertheless, we decided to show the results for all the programs. We observed that the programs with a higher number of faults show a similar ODC fault distribution; the only observed deviation was presented by "Joe" program, which had more checking faults than the global trend. This trend existing across programs reinforces the suggestion that software faults do follow a clear pattern of distribution across ODC types.

11.2.3 Extended Classification and Discussion

For the purpose of fault injection the fault types provided by ODC are not practical as they are too broad, meaning that many different faults fall in the same type and the types themselves lack the fine details required by an automated tool to be able to reproduce the fault in the target code. Clearly, further refining is needed, not in the sense of an alternative classification but as an additional detail layer to ODC. As explained in Sect. 11.2.1, we propose to achieve this extra layer by analyzing faults from the point of view of the (program) context in which fault occur and relate the faults with programming language constructs. Using this notion, a defect is then one or more programming language constructs that are either missing, wrong or in excess. A construct is any building block of the traditional programming languages:

Table 11.3 Fault distribution across ODC types by individual programs

Programs	CDEX	Vim	FCiv	Pdf2h	GAIM	Joe	ZSNES	Bash	LKernel	FireBird	MingW	M	Total (%)
# faults	11	249	53	20	23	78	3	2	93	2	60	74	668
ODC type													
Assignment (%)	18.2	21.3	11.3	55	4.3	25.6	66.7	100	22.6	50	10	24.3	21.4
Checking (%)	18.2	22.5	13.2	5	52.2	44.9	0	0	25.8	50	38.3	8.1	25
Interface (%)	54.5	6.4	7.5	0	4.3	14.1	0	0	5.4	0	5	4.1	7.3
Algorithm (%)	9.1	44.6	52.8	40	26.1	15.4	33.3	0	33.3	0	46.6	56.8	40.1
Function (%)	0	5.2	15.1	0	13	0	0	0	12.9	0	0	6.8	6.1

statements, expressions, function calls, etc. Following this idea, we classified each fault according to its nature which can be one of these: missing construct, Wrong construct, or Extraneous construct. Although this classification is orthogonal to ODC and can be used alone (as is in Table 11.4), we used it as an extension to ODC fault types to provide a refined view of the faults specifically aimed at emulation by fault injection.

As we can see in Table 11.4, faults of the extraneous nature are clearly less frequent than the other two. This was an expected result, as programmers are more prone to forget to put something in the program, or to put it in a wrong way, than to insert surplus code. We can also see that missing programming constructs seem to be the dominant type of software fault. From the point of view of representativeness for fault injection experiments, this information is valuable.

Table 11.5 presents the total number of missing, wrong or extraneous faults for each of the five ODC fault types addressed in this study. We also provide some examples of fault to help the reader understand what kind of fault is included in each type (this will be detailed further on). As we can see from Table 11.5, there are once again trends that we can use to achieve representativeness in the injection of software faults, e.g., for the assignment and interface types, missing program construct faults are less frequent than the wrong construct faults.

We then further detailed the description of faults describing exactly what constructs were missing, wrong or extraneous. We did this for all ODC types and obtained a reasonable small list of fault types (for each ODC type). This is an interesting result, as we do not want a small list of generically-described faults where many faults fit and for which no practical tool can emulate those faults due to lack of details, and we also do not want a long list of over-detailed description where each fault fits into and only into its own type, rendering any effort of representativeness useless. The complete list of fault types for all ODC types is outside the goal of this section and chapter. We present here in Table 11.6 the list of faults for the ODC type algorithm and refer the user to [315] for a detailed description of this work.

The faults listed in Table 11.6 are now described with a level of detail that is useful for practical fault injection. For example, the type MFC—missing function call refers to the omission of a call to a routine in the program. This is an easy understandable description that can be easily emulated into the target code. Another important issue is the identification of suitable location where a given fault can be injected. Using the MFC fault type again, it is relatively easily to identify occurrences of function call in the target, even in the binary code. It is worth noting that this study was part of an effort to devise and implement a fault injection technique able to inject realistic software fault directly into the binary code of the target, without requiring source code (goal that was achieved). This scenario is relevant because many fault injection applications involve common-of-the-shelf components for which there is no source code available.

To help readers understand the level of details that is now used to describe faults, we use another example from Table 11.6 . Fault MIFS—Missing if construct plus statements. This fault refers to the omission of a conditional statement deciding if a givel (small) block of statements is executed. In C language it is something like

Table 11.4 Fault distribution by fault nature

Fault nature	CDEX	Vim	FCiv	Pdf2h	GAIM	Joe	ZSNES	Bash	LKernel	Firebird	MingW	ScumVM	Total	(%)
Missing construct	3	157	35	11	17	34	1	0	63	2	45	61	429	(64.2)
Wrong construct	8	85	18	9	6	41	2	2	24	0	14	12	221	(33.1)
Extraneous construct	0	7	0	0	0	3	0	0	6	0	1	1	18	(2.7)

Table 11.5 Fault nature totals across ODC types

ODCtype	Nature	Examples	# faults	% of total
Assignm.	Missing	A variable was not assigned a value, a variable was not initialized	62	9.3
	Wrong	A wrong value (or expression result, etc) was assigned to a variable	70	10.5
	Extraneous	A variable should not have been subject of an assignment	11	1.6
Checking	Missing	An "if" construct is missing, part of a logical condition is missing,etc	113	16.9
	Wrong	Wrong logical expression used in a condition in brach and loop on struct (if, while, etc.)	53	7.9
	Extraneous	An "if" construct is superfluous and should not be present	1	0.1
Interface	Missing	A parameter in a function call was missing; incomplete expression was used as param.	11	1.6
	Wrong	Wrong information was passed to a function call (value, expression result etc)	38	5.7
	Extraneous	Surplus data is passed to a function (e.g. one parameter too many in function call)	0	0
Algorithm	Missing	Some part of the algorithm is missing (e.g. function call, a iteration construct, etc)	222	33.2
	Wrong	Algorithm is wrongly coded or ill-formed	40	6
	Extraneous	The algorithm has surplus steps; A function was being called	6	0.9
Function	Missing	New program modules were required	21	3.1
	Wrong	The code structure has to be redefined to correct functionality	20	3
	Extraneous	Portions of code were completely superfluous	0	0

If (cond) {statement1; statement2; …. }

Once again the identification of this type of construct is easily identifiable in the target code and easily emulated through modification in said code. One very important aspect of the information in Table 11.6 is the number of occurrences for each fault type. The two fault types described here are much more common than other types (e.g., MIEA). This is a very important information to build representative faultloads for fault injection experiments. Table 11.7 presents a global view of all the occurrences for all fault types (all ODC types and programs).

The information summarized in Table 11.7 is very relevant. It offers two conclusions about software faults:

Table 11.6 Detailed analysis of algorithm faults

Fault nature	Fault specific types	CDEX	Vim	FCiv	pdf2h	GAIM	Joe	ZSNES	Bash	LKernel	FireBird	MinGW	Scum	VM	Total
Missing construct	Missing function call (MFC)		28	7	1	1	5			4		2	23		71
	Missing If construct plus statements (MIFS)		27	10			1			15		15	12		80
	Missing if-else construct plus statements (MIES)											4	3		7
	Missing if construct plus statements plus else before statements (MIEB)	1	10	4		2						1			18
	Missing if construct plus else plus statements around statements (MIEA)									2		1			3
	Missing iteration construct around statement(s) (MCA)		1												1
	Missing case: statement(s) inside a switch construct (MCS)		1												1
	Missing break in case (MBC)		3				1								4
	Missing small and localized part of the algorithm (MLPA)		9		4	2	1			1		5	1		23
	Missing sparsely spaced parts of the algorithm (MLPS)		5		1										6
	Missing large part of the algorithm (MLPL)		3			1		1		3					8
Wrong construct	Wrong function called with same parameters (WFCS)			1			2			6					9
	Wrong function called with different parameters (WFCD)		9	1									3		13
	Wrong branch construct—goto instead break (WBC1)		1	1											2
	Wrong algorithm—small sparse modifications (WALD)		4	1	1										6
	Wrong algorithm—code was misplaced(WALR)		5	3	1										9
	Wrong conditional compilation definitions (WSUC)		1												1
Extraneous	Extraneous function call (EFC)		4				2								6
Total faults found		1	##	28	8	6	12	1	0	31	0	28	42		268

Table 11.7 The "Top-N" fault in this study by occurrence frequency

Fault types	Description	Fault coverage (%)	ODC types
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	Assignment
WLEC	Wrong logical expression used as branch condition	3	Checking
WVAV	Wrong value assigned to a value	2.44	Assignment
MVAV	Missing variable assignment using a value	2.25	Assignment
MVI	Missing variable initialization	2.25	Assignment
WAEP	Wrong arithmetic expression used in parameter of function call	2.25	Interface
WPFV	Wrong variable used in parameter of function call	1.50	Interface
	Total faults coverage (field data)	50.69	

- There is a relatively small set of fault types that is responsible for a large portion of all the fault occurrences. The 12 fault types in Table 11.7 put together are responsible for 50 % of all the faults discovered in this field study.
- There are faults that are clearly more frequent than others, and this information is important to build representative faultloads for fault injection scenarios.

The results of this field study are very interesting for research on software faults and for the injection of software faults. It offers insight on fault details aimed at the realistic emulation of faults, it offers information about the distribution of the most common type of faults in the operational scenario aimed at generating representative faultloads, and is the basis of the G-SWFIT technique for fault injection. These results and this technique have been used on several research works (e.g., [315, 664, 665]), and the classification scheme is used as basis for different application areas (still related to software faults), such as security (e.g., relate vulnerabilities with its root cause faults).

To conclude the presentation of this field study we present here one example of a software fault as classified and described in this field study (Fig. 11.1), and one example of a fault emulation operator of the G-SWFIT fault injection technique developed in the sequence of this field study (Fig. 11.2). We refer the reader to [315] for more details.

```
*** 4858,4864 ****
      for (p=name; isalpha(*p) || isdigit(*p) || *p == '_'; ++p)
         ;
!     if (p == name)
      {
         EMSG("Function name required");
         return;
      }
--- 4858,4864 ----
      name = eap->arg;
      for (p=name; isalpha(*p) || isdigit(*p) || *p == '_'; ++p)
         ;
!     if (p == name && !eap->skip)
      {
         EMSG("Function name required");
         return;
      }
```

Was missing

Fig. 11.1 Example of a diff/patch file (excerpt). In this example, the patch applies a "&& !eap->skip" that was missing. The fault type is MLAC—Missing "AND EXPR" in expression used as branch cond

Operator	Example	Example with fault	Search pattern	Code change

OMIEB	if (*expression*)
	{
	statements-IF
	}
	else
	{
	statements-ELSE
	}
	
	... remaining code
		~~if (expression)~~
		~~{~~
		~~*statements-IF*~~
		}~~
		~~else~~
		~~{~~
		statements-ELSE
		~~}~~
		
		... remaining code
			flag-affecting instr.
			jcond elsecode
			... instrs (IF)
			
			jmp *after*
			elsecode:
			... instrs (ELSE)
			
			after:
			... remaining code
				- All the conditional jumps to the address loc01 are changed into unconditional jumps
				- Call instructions and stores to memory existing between the cond jumps are removed

Notes
There may be several cond. jumps to *elsecode* if expressions is composed of several sub-expressions
The side-effects (if any) of the first sub-expression are not ommited

Fig. 11.2 Operator to emulate a fault OMIEB—missing if construct and the statements surrounded by it plus an else statement. It is not one of the most common fault types, but it serves to illustrate the changes at the high level code and its related modification at low level to emulate the fault, as well as search pattern used to identify suitable fault locations

11.2.4 Considerations on the Case Study

In this case study a large number of software faults were analyzed to improve the knowledge about the nature of software faults: its nature, the frequency of its occurrence frequency by fault types, and how they can be emulated through fault injection. The contributions of this case study were a fault classification scheme allowing

practical injection of software faults and the knowledge about the fault distribution across fault type as they occur in the operational scenario. The source of the data was a set of open-source programs, without which this study would have been much harder if not impossible: in closed-source projects, the information regarding faults and their correction is kept within the development team. As the correction of faults (patch code) was directly used to conduct this field study, we stress the importance of having data available for research purposes, even in closed-source projects. This data can hardly be used for commercial purposes, and, excepting issues related to security, a concerted effort should be made by academia to try and obtain data such as the one used for this study. This effort should be articulated with the creation of data repositories to help spreading the data and results of field data studies.

11.3 Case Study 2: Field data on Security Vulnerabilities

In this section we present the results of a field study on the most common vulnerabilities, which provides a truthful body of knowledge on real security vulnerabilities that accurately emulate real world security problems. The data was obtained by analyzing past versions of representative web applications with known vulnerabilities that have already been corrected. The main idea is to compare the piece of defective code with the corrections made to secure it. This code change (or the lack of it in the vulnerable application) can be viewed as the reason for the presence of the vulnerability. Note that this methodology can generically be used in other field studies to obtain the characterization and distribution of the source code defects that originate vulnerabilities in web applications.

The field study uses data from 655 SQL Injection and XSS security patches of six widely used web applications. The detailed analysis of the code of the patches shows that web application vulnerabilities result from software bugs affecting only a restricted collection of statements, which greatly facilitates the emulation of vulnerabilities through fault injection, as the effort can be concentrated on the emulation of vulnerabilities in a small number of types of statements.

Sections 11.3.1 and 11.3.2 describe the methodology used to collect the field data in this field study. Section 11.3.3 presents the systems addressed in the study, and the vulnerabilities addressed are presented in Sect. 11.3.4. Section 11.3.5 details the information gathered in the study and the results are presented in Sect. 11.3.6. Section 11.3.7 summarizes this case study.

11.3.1 Vulnerability Analysis and Classification Methodology

When web application vulnerabilities are discovered, software developers correct the problem releasing application updates or patches. In our study, we used these patches to understand which code is responsible for security problems in web applications.

With this approach, we can classify the code structures that cause real security flaws and identify the most frequent types of vulnerabilities observed in the web applications considered in our field study.

For each web application under test, the methodology to classify the security patches is the following:

1. Verification of the patch to obtain the right version of the web application where it applies. We need to confirm the availability of the specific version of the web application and obtain it for the rest of the process. It is mandatory to have both the patch and the vulnerable source code to be able to analyze what code was fixed and how, unless the patch file has all this information (which we found to be unusual).
2. Analysis of the code with the vulnerability and compare it with the code after being patched. The difference between the vulnerable and the secure piece of code is what is needed to correct the vulnerability. This is what the software developer should have done when he first wrote the program and this is what we have to classify.
3. Classification of each code fix that is found in the patch. The absence of the actions programmed in the patch represents what causes the vulnerability. For example, if the patch replaces the variable $id with intval($id)[1], we consider that the vulnerability is caused by the absence of the intval function in the original code. To be accurate, we followed the patch code analysis guidelines described in the next section.
4. Loop through the previous steps until all available patches of the web application have been analyzed.

11.3.2 Patch Code Analysis Guidelines

Web applications are developed using different coding practices and during the classification of the security patches we face different scenarios and have to make some decisions that need to be clarified. To avoid classification mistakes and misinterpretations the following guidelines are followed:

1. We assume that the information publicly disclosed in specialized sites is accurate and that the fix developed by the programmer of the patch and made available by the company that supports the web application solved the stated problem. We do not test the presence of the vulnerability nor confirm its correction.
2. To correct a single vulnerability several code changes may be necessary. This way, each code change was considered as a singular fix. For example, suppose that two functions are needed to properly sanitize a variable. Missing any of these functions makes the application vulnerable, so both of them must be taken into

[1] The intval is a PHP function that returns the numeric value of a variable, or 0 on error.

account. In this case, if we want to simulate the vulnerability, we may remove any of the singular fault type fixes.

3. When a patch can fix several vulnerability types simultaneously, each one is accounted separately. This occurred naturally because we analyzed each vulnerability independently, as if we were doing several unrelated analyses, one for each vulnerability type. For example, this occurs when a not properly sanitized variable is used in a query (e.g. allowing SQL Injection) and later on is displayed on the screen (e.g. allowing XSS). When this variable is properly sanitized, both vulnerabilities are mitigated simultaneously, however this situation accounts for the statistics of both XSS and SQL Injection vulnerabilities.
4. When a particular code change corrects several vulnerabilities of the same type, each one is considered as a singular fix. For example, suppose that the value assigned to a specific variable comes from two sources of external inputs; and the variable is displayed in one place without ever being sanitized. We consider that the application has two security vulnerabilities because it can be attacked from two different inputs. However, to correct the problem all that is needed is to sanitize the variable just before it is displayed. In this example we consider that two security problems have been fixed, although only one code change was needed.
5. A security vulnerability may affect several versions of the application. This happens when the code is not changed for a long time, but it is vulnerable. The patch to fix the problem is the same for all versions, and therefore it is considered to be only one fix.

By following the previous guidelines, it was possible to classify almost all the code fixes analyzed. However, in some situations, patching one or more vulnerabilities may involve so many changes, including the creation of new functions or a change in the structure of the overall piece of code, that it is too difficult to classify it properly. These situations are usually associated with major code changes involving simultaneously security and other bug fixes related to functional aspects. These occurrences were quite marginal (5.4 %) and were not considered in our study because they are too complex and difficult to analyze due to the lack of source code documentation.

11.3.3 Web Applications Analyzed

One mandatory condition for our field study is to have access to the source code of the web applications under analysis. The code of previous versions and the associated security patches must also be accessible. The other mandatory condition is the availability of information correlating the security fix and the specific version of the web application.

The goal is to be sure that it is possible to access the source code (including the code of older versions) in order to be able to analyze and understand the security

vulnerability and how it was fixed. Actually, the way a given vulnerability is fixed is a key aspect in the classification of the fault type originating the vulnerability.

For the present study we have selected six LAMP (Linux, Apache, MySQL and PHP) web applications: PHP-Nuke [726], Drupal [307], PHP-Fusion [493] , WordPress [943], phpMyAdmin [728] and phpBB [727]. These are open source web applications that represent a large community of users and, fortunately, there is enough information available about them to be researched. Additionally, they represent a large slice of the web application market and have a large community of users:

- Drupal (winner of the first place at the 2007 and 2008 Open Source CMS Award), PHP-Fusion (one of the five winner finalists at the 2007 Open Source CMS Award) and phpBB (the most widely used Open Source forum solution and the winner of the 2007 SourceForge Community Choice Awards for Best Project for Communications) are Web Content Management Systems (CMS). A CMS is an application that allows an individual or a community of users to easily create and administrate web sites that publish a variety of contents.
- PHP-Nuke is a well-known web based news automation system built as a community portal. PHP-Nuke is one of the most notorious CMS and it has been downloaded from the official site over 8 and half million times.
- WordPress is a personal blog publishing platform that also supports the creation of easy to administrate web sites. It is one of the most used blog platforms in the World.
- phpMyAdmin is a web based MySQL administration tool. It is one of the most popular PHP applications, is included in many Linux distributions, and was the winner of the 2007 SourceForge Community Choice Awards for Best Tool or Utility for SysAdmins.

The six web applications analyzed are so broadly used since several years ago that they have a large number of vulnerabilities disclosed from previous versions, which were the subject of analysis of the field study. It is important to emphasize that a single vulnerability opens a door for hackers to successfully attack any of the millions of web sites developed with a specific version of the web application. Furthermore, it is common to find a single vulnerability in a specific version that also affects a large number of previous versions. The overall situation is even worse because web site administrators do not always update the software in due time when new patches and releases are available.

11.3.4 Security Vulnerabilities Studied

In the present work we focus on two of the most critical vulnerabilities in web applications: XSS and SQL Injection. A Cross Site scripting (XSS, but also known as CSS) vulnerability allows the attacker to inject HTML and/or a scripting language (usually JavaScript) into a vulnerable web page [710] . A SQL Injection vulnerability

allows the attacker to tweak the input fields of the web page in order to alter the query sent to the back-end database [709].

Exploits of these vulnerabilities take advantage of unchecked input fields at user interface, which allows the attacker to change the SQL commands that are sent to the database server (SQL Injection), or allows the attacker to input HTML and a scripting language (XSS). Two main points account for the popularity of these attacks:

- The easiness in finding and exploiting such vulnerabilities. They are very common in web applications and within a web browser the attacker can probe for these vulnerabilities tweaking GET and POST variables that are available in the HTML page. The building of an exploit for fun or profit can be a bit more time consuming, but there are plenty information and guides on how to do it (e.g. look at [409, 708] for XSS and [408, 708, 720] for SQL Injection, just to mention a few).
- The importance of the assets they can disclose and the level of damage they may inflict. In fact, SQL Injection and XSS allow attackers to access unauthorized data (read, insert, change or delete), gain access to privileged database accounts, impersonate another user (such as the administrator), mimicry web applications, deface web pages, get access to the web server, malware injection, etc. [347].

11.3.5 Patch Code Sources

For all the applications analyzed, we collected the source code of both the vulnerable and the patched versions. By comparing these two versions, we could understand the characteristics of the vulnerability and classify what code was changed to correct it.

Software houses and developers follow their own policies in what concerns the public availability of older versions of the software, particularly when they have security problems. In some cases, they can be hard to find and even the access to the past collection of vulnerability patches can be a cumbersome task. Furthermore, most security announcements publicly available are so vague that it is too difficult (or even impossible) to know which source files of the application are affected by a particular vulnerability. Moreover, some of the disclosed information about security problems is too generic and groups together several types of security vulnerabilities (e.g., using the same document to refer to directory traversal, remote file inclusion and COOKIE poisoning vulnerabilities), which makes it more difficult to map our target vulnerabilities to the code fixing them.

In order to gather the actual code of security patches, we have to use several sources of data, such as mirror web sites, other sites that provide the source code (mainly on blogs or forums), online reviews, news sites, sites related to security, hacker sites, change log files of the application, the version control system repository, etc.

For the purpose of this study, we just need the changes made to the code of the application correcting the vulnerability problem (i.e., the source code of the entire application is not required). However, as there is no standard way of providing the data about the security vulnerability fix, different sources of information have to be

considered, each one following its own specific format. The four main source types used in the current work are the following:

1. Security patch files with information about the target version of the application. In this case, we have the reference to the buggy version of the web application and to the patch file that must be applied to mitigate the target vulnerability.
2. Updated version of the web application. Actually, this is a completely new version of the application containing new features and bug fixes (including security ones). This is the most common source of information we have found, but it is also the one that needs more exploration work to be done.
3. Available security diff file. In this case, there is a diff file, which is a file containing only the code differences between two other files with information about what lines of the original file have been removed, added or changed. It has, therefore, the precise code changes needed to fix a referenced vulnerability.
4. Version control system repository. Almost all relevant open source applications are developed using a version control system to administer the contributions of the large community of developers from around the world. This is the most complete source of information we can have about the application, although it may be difficult to find what we are looking for in such a vast collection of files and versions.

Once the vulnerable code and the respective patch are obtained using one of the previous sources of information, a differential analysis is performed to identify the locations in the code where the defects are fixed. This operation is done mainly through the use of diff utility. The Unix diff utility is a file comparison tool that highlights the differences between two files using the algorithm to solve the longest common subsequence problem [455]. A manual analysis of the code can be also performed when the output of the diff utility is too complex due to a large number of changes between the two versions of the source code, or when many corrections are done in the same file. The manual analysis also helps grouping several security corrections and discarding the code changes not related to security issues.

11.3.6 Field Study Results and Discussion

In the field study we classified 655 XSS and SQL Injection security fixes found in the six web applications analyzed (PHP-Nuke, Drupal, PHP-Fusion, WordPress, phpMyAdmin and phpBB). We followed a classification scheme based on the software fault classification proposed in [314] and adapted the fault types specific to XSS/SQL injection (e.g., MFC to MFCext).

The overall distribution of the fault types found in the six web applications analyzed is shown in Table 11.8. In this table we can see the individual results for each fault type allowing us to understand how they are distributed along the web applications analyzed.

A common belief is that vulnerabilities related to input validation are mainly due to missing if constructs or even missing conditions in the if construct. However, our

Table 11.8 Detailed results of the field study on the most common software faults generating vulnerabilities

Web app.	PHP-Nuke		Drupal		PHP-Fusion		WordPress		phpMyAdmin		phpBB		
Fault type	SQL	XSS	SQL	XSS	SQL	XSS	SQL	XSS	SQL	XSS	SQL	XSS	%
MFCext.	120	133	4	39	6	13	6	94	1	51	3	27	76
WPFV	31			3	2	5				4		1	7
MIFS	5	2		2	7	6				10		2	5
WVAV	2			3				2		4		17	4
EFC					1					1		4	1
WFCS				3	1	1		13					3
MVIV		1			1	3						4	1
MLAC				1	2	4				2			1
MFC				2	1					1			1
MIA				1		1							0
MLOC		1											0
ELOC				1									0
Total faults	158	137	4	55	21	33	6	109	1	73	3	55	100

field study shows that this is not the case, as the overall "missing IF…" fault types (MIFS and MIA: see Table 11.8) only have a weight of 5.5 %. As for the "missing <condition>…" fault types (MLAC and MLOC), they represent only 1.52 % of all the fault types. This suggests that programmers typically do not use if constructs to validate the input data, and this may occur due to the complexity of the validation procedures needed to avoid XSS and SQL Injection.

The typical approach we found in the field is the use of a function to clean the input data and let it go through, instead of stopping the program and raise an exception (or show an error page). This may be understood as a design goal trying to prevent the disruption of the interaction of users to the least possible. In what concerns security, it would be better to allow only inputs known as correct (white list) as this prevents any input with suspicious characters to go any further and is more secure than just cleaning the input from malicious characters and let the operation continue normally.

Analyzing the global distribution of web applications vulnerabilities we found 70.53 % of XSS and 29.47 % of SQL Injection showing that XSS is the most frequent type by far. As shown, all the fault types account for XSS vulnerabilities but only eight fault types report to SQL Injection, which might help justify the fact that XSS is more prevalent than SQL Injection, confirming the results of the IBM X-Force®2008 Trend and Risk Report [819]. This trend is also confirmed by vulnerability reports disclosed in CVE [657, 707]. However, the four fault types that do not contribute to SQL Injection (MFC, MIA, MLOC and ELOC) only account for 1.22 % of all the fault types. Obviously, we do not have enough sample values to conclude that SQL Injection may not be derived from one of these fault types. We can only say that we did not find them in our field study.

There are several factors that contribute to the prevalence of XSS. XSS is easier to discover because it manifests directly in the tester web browser window. Every

input variable of the application is a potential attack entry point for XSS, which is not the case for SQL Injection, where only variables used in SQL queries matter. Another factor that contributes to the prevalence of XSS is that SQL Injection alters the database records and this cannot be always seen in the interface, at least so explicitly as XSS. Moreover, the knowledge needed to test for XSS [409, 708] is not as complex as for SQL Injection, for which the attacker needs to have deep knowledge about the SQL language. Although the SQL language is usually based on the SQL-92 standard [290], every database management system (DBMS) has its own extensions and particularities [408, 708, 720], that need to be taken into account when searching for SQL Injection.

The most representative and widespread fault type is the "Missing function call extended (MFCext.)". It represents 75.87 % (140 SQL Injection + 357 XSS out of 655 vulnerabilities studied) of all the fault types found. The high value observed for the MFCext fault type comes from the massive use of specific functions to validate or clean data that comes from the outside of the application (user inputs, database records, files, etc.). In many cases, functions are also used to cast a variable to a numeric value, therefore preventing string injection in numeric fields.

The next three most common fault types are "wrong variable used in parameter of function call (WPFV)", "missing IF construct plus statements (MIFS)", and "wrong value assigned to variable (WVAV)".

A recurring problem is that, looking at several versions of the same program, we frequently found the same regex string being slightly updated as new attacks are discovered. These situations were found in WPFV and WVAV faults.

Excluding the faults types already discussed (MFCext., WPFV, MIFS and WVAV), the remaining fault types correspond to only 7.63 % of the security vulnerabilities found. These fault types are EFC, WFCS, MVIV, MLAC, MFC, MIA, MLOC and ELOC.

11.3.7 Considerations on the Case Study

In this case study we presented a methodology for characterizing the most frequent fault types associated with the most common web application vulnerabilities based on a field study. We focused on XSS and SQL Injection vulnerabilities of six widely used web applications, using 655 security fixes as the field data. Results show that only a small subset of 12 generic software faults is responsible for all the XSS and SQL Injection vulnerabilities analyzed.

One relevant outcome of the field study performed is referred to the distribution of vulnerabilities by a reduced number of fault types. In fact, we observed that a single fault type, the MFCext. (missing the function responsible for cleaning the input variable), is responsible for about 76 % of all the security problems analyzed. Previous studies on software fault types [212, 312] also show this large dependency on a few bug types. Furthermore, this trend is not new in the security area: Microsoft has already stated that fixing the top 20 % of the reported bugs eliminates around 80 % of errors [785] and the Gartner Group reported that 20 % of security test rules uncover

80 % of errors [574]. This concentration of the responsibility of most vulnerabilities on just a few fault types can be very important to address the web applications security and makes it feasible to emulate vulnerabilities by means of fault injection, which has already been started to be addressed by the research community [322, 342, 344, 815].

11.4 Overview of Data Repositories

Data repositories are an excellent resource to store and share information for research purposes. One type of valuable information that can be shared through data repositories is the result from field data studies. Although data repositories to store failure data and dependability experiments results are relatively rare (especially considering the huge value of real failure data to help designers in improving computer systems), several initiatives have been proposed and are currently available.

The Data & Analysis Center for Software (DACS) is a Department of the US Defense Information Center supporting research on software reliability and quality. It serves as centralized source for data related to software metrics. The DACS maintains the Software Life Cycle Experience Database (SLED). This repository is intended to support the improvement of the software development process. The SLED is organized into nine data sets covering all phases and aspects of the software lifecycle ([253] and [477]). Examples of these datasets are:

- The DACS Productivity Dataset (collected from government and private industry sources). This dataset consists of data on over 500 software projects and is mainly oriented to software cost modelling and productivity analysis [678]. The data represents software from early 60s to early 80s and includes software projects ranging from avionics to off-the-shelf packages. The information in this dataset includes the following: size of project, effort, language, schedule, errors.
- The NASA/SEL Dataset (contributed by the Software Engineering Laboratory (SEL) at NASA Goddard Space Flight Center). This repository maintains data on avionic applications since 1976. The dataset is available by request on disk and it can be accessed through web browser. Using the latter, users have access to analytical summaries including linear regression, scatter plots and histograms. The analytical results are created dynamically per request during the HTTP session and served to the user browser. The repository information is stored in a relational database and the link between the data repository and the web server is supported through Perl applications.
- The Software Reliability Dataset (collected at Bell Laboratories) [669]. This repository describes failures in a wide range of application domains including real time, control, office, and military applications. This dataset was primarily aimed at the validation of software reliability models and to assist software managers to monitor and predict software tests. As in the NASA/SEL dataset, the information can be obtained by request, and it can also be accessed through web interface.

The Metrics Data Program (MDP) Repository is a database maintained by the NASA Independent Verification and Validation facility [674]. The repository is aimed at the dissemination of non-specific data to the software community and it is made available to the general public at no cost. All the data available in the repository are sanitized by the projects representatives, and all the necessary clearances are provided. Users of the repository are free to analyze the data for their specific research goals.

The MDP repository is part of the MDP on-going effort to improve the ability to predict error in software by improving the quality of the problem data related to software (e.g., improve the quality of the information about the relationship of the error and the development phase). To this effort, the MDP recruits the participations of private-sector and public-sector projects. Recruited projects maintain complete control of data release and the level of participation in the program. The effort required by the participating projects is minimal. The repository contains data on the software projects that were collected and validated by the MDP program, spanning more than 8 years and including more than 2700 error reports. The information stored in the repository consists of error data, software metrics data, and error data at the function/method level. The dataset enables data associations between products, metrics, and errors classified according to the Orthogonal Defect Classification (ODC) [204].

The Software Reference Fault and Failure Data Project [689] is maintained by the National Institute of Standards and Technology and is aimed at the development of metrology, taxonomy and repository for reference data for software assurance. The project maintains a repository on software fault data specifically aimed at helping industry protect against releasing software systems with faults and to help assess software systems quality by providing statistical methods and tools. The repository is available to the public upon request. The access to the information online allows users to view data and execute simple queries. Analytical and statistical use of the data is possible through a program developed within the project and available to the public (the EFFTool).

The Computer Failure Data Repository (CFDR) is a public repository on computer failure data ([74] and [182]) supported by USENIX. The repository is aimed at the acceleration of the research on system reliability with the ultimate goal of reducing or avoiding downtime in computer systems. To this goal, the CDFR hopes to remove the main difficulty faced by researchers, which is the lack of reliable and precise information about computer failures. The CDFR repository is open to both obtaining and contributing data. The repository comprises nine independent data-sets focusing mainly on very large storage systems. The repository information covers many aspects, including: software failures, hardware failures, operator errors, network failures, and operational environment problems. The raw data are available to the public [182] through web interface. The project does not offer online capability for analytic and statistical data-processing.

The AMBER Raw Data Repository [32] is a repository of field data and raw results from resilience assessment experiments. Its goal is to grant both the research and IT industry communities with an infrastructure to gather, analyze and share field data resulting from resilience assessments of systems and services, stimulating a better coordination of high quality research in the area, and contributing to the promotion

of a standardization of resilience measurement, which will in turn have a positive impact in the industry. While experimental and field data repositories are recognizably fundamental for supporting the advance of research and the dissemination of knowledge, the research community still seems somewhat reluctant in embracing such enterprises. This repository aims to encourage acceptance from the community to share its data and promote the research involving several partners sharing data.

Publicly available vulnerability databases currently play a very important role in making the information on vulnerabilities available to researchers and have completely reshaped the way software vulnerabilities are reported and disseminated in recent years. Examples of popular vulnerability databases are the National Vulnerability Database [693] and The Open Source Vulnerability Database [705], which provide comprehensive reports about discovered software vulnerabilities including the nature of a vulnerability (its type, the component where it was located, the list of vulnerable system versions, its discovery date, and so on) and include examples on how to exploit it, as well as the patch or the workaround provided by system vendor to fix it (when available). Additionally, to alert users about the severity and security risk of reported vulnerabilities, these databases typically provide vulnerability impact and exploitability levels assigned by security advisors. These databases also provide a web-based interface that enables users to search vulnerabilities and browse a list of the vulnerabilities reported for a given system.

11.5 Conclusion

The case studies presented in the chapter allow drawing some conclusions on field measurements and field data studies. Although the focus of the chapter is software faults and security vulnerabilities, these conclusions apply to any type of measurement obtainable in the field. Important aspects that are self-evident are the representativeness of the measurements and results, the classification used to describe them and manipulate data, and the mechanisms to make data and results available to the research community and general public.

Concerning data on the robustness of the computer-based systems, field data is mostly obtained from reports (bug reports, incident reports, security logs, and so on, depending on the nature of the incident). These reports are filed by the users and operators and are typically used by the system developers to solve the incidents and improve the system.

Observations made in closed-source, proprietary systems are typically not available to the public. Observations originating from open-source systems are normally made available to the community (e.g., stored in a repository). However, these repositories are normally not oriented to a systematic storage and classification of the discovered faults and remedies. Instead they are the result of the accumulation of solution to problems resulting in a kind of logfile-like information about which problems were discovered (bug reports, many times repeated), and how were solved. The exception to this are the repositories maintained by researchers in the context of

long-term research in large companies, such as IBM. These are good initiatives, but typically are very different from one another. It would be of great value to the research community to have information on software faults available in a systematized and uniform way. Repositories like the ones described in the chapter are good initiatives in that direction.

Concerning security, the information pertinent to research is even harder to find than those about software faults. It is not the case of data availability (as it is for faults in closed-source systems). On the contrary, there is plenty of information. The major problem is that there is too much information, scattered and mostly repeated, and classified using different schemes. A given security issue may have been classified according to in scheme and given one value of severity, for instance, and in another repository, the same vulnerability may appear with a different description and different characterization.

The usefulness of public repositories to the research communities is demonstrated by the existence of studies based on the information stored in publicly available repositories (e.g. [32]). Nevertheless, and in spite of the different repository initiatives already available, the raw data from the vast majority of research works on experimental dependability evaluation and on field failure data, among other examples, is not available in any repository. Hundreds of papers have been published but the raw data that have led to the final results presented in those papers is not available. Data repositories do seem a very promising initiative to provide the means to have a uniform description of raw data and results and make this information available to the public, and perhaps some more concerted effort should be placed towards creating and maintaining said repositories. One example among several is the AMBER repository, which was built specifically to share data among different teams.