Computing manuscript
http://link.springer.com/article/10.1007/s00607-018-0657-y

Emulating representative software vulnerabilities using field data

Raul Barbosa · Frederico Cerveira · Luís Gonçalo · Henrique Madeira

This is a pre-print of an article published in Springer Computing. The final authenticated version is available online at: https://doi.org/10.1007/s00607-018-0657-y

Abstract Security vulnerabilities are a concern in systems and software exposed via networked interfaces. Previous research has shown that only a minority of vulnerabilities can be emulated through software fault injection techniques. This paper aims to accurately emulate software security vulnerabilities. To this end, the paper provides a field-data study on the operators needed to emulate vulnerabilities in software written in the C programming language. A practical implementation is constructed and the feasibility of emulating software vulnerabilities is evaluated. The emulation operators were obtained by analyzing publicly available vulnerability databases for the Linux kernel, the Xen hypervisor, and the OpenSSH tool. The results show that a typical security vulnerability involves a single function and consists of combinations of up to three fault operator instances. The expected impact of this study is to allow practical emulation of security defects in large software projects, to support software quality and security assessment.

Keywords Security · Dependability · Security vulnerabilities · Software faults

1 Introduction

Software vulnerabilities are a major concern for developers and organizations of all kinds, as they represent serious security risks to computer systems, especially those allowing remote access via networked interfaces. The increasing size and complexity of software projects lead to a proliferation of software defects,

R. Barbosa
CISUC, Department of Informatics Engineering
University of Coimbra
P-3030 290, Coimbra, Portugal
Tel.: +351 239 790 024
Fax: +351 239 701 266
E-mail: rbarbosa@dei.uc.pt

informally known as bugs. While most software bugs create no security risks, a subset of those faults opens weaknesses that can be exploited to compromise a system's information security. Furthermore, as software vulnerabilities do not change the functionalities of the software, they are invisible to most of the current testing strategies, requiring specific static and dynamic approaches. Thus, managing a software project's security bugs is both an essential task and a complex challenge.

This paper addresses the emulation of software security defects for the purpose of software quality and security assessment. Software faults have been widely studied by the software reliability community, namely through detailed analysis and classification of real defects found in production software [5] and, subsequently, studies on the nature of software faults in open-source projects aiming to emulate software defects introduced by developers [9]. However, security defects have received much less attention and, to the best of our knowledge, no general approach has been proposed thus far to emulate realistic security bugs in the C programming language, which is widely used for building systems software.

The first step toward emulating realistic software vulnerabilities is to understand the most common programming mistakes that result in security vulnerabilities. To this end, public databases containing Common Vulnerabilities and Exposures (CVEs) on software projects provide rich data on which we base our field study. We used the Orthogonal Defect Classification method [4] to classify and analyze the programming mistakes (vulnerabilities) helping us to clarify *what* a security bug is. Understanding *how* to emulate security defects then involves combining well-known software fault injection techniques with the specific instances of programming mistakes found in real software projects.

In general, vulnerability injection consists in deliberately inserting security defects in a system, with the goals of software quality and security assessment. One may identify three main usage scenarios as follows:

- *Validation of vulnerability scanners and source code analysis tools.* Computer systems and applications exposed to security risks are the object of numerous techniques aiming to detect, isolate and recover from attacks. Vulnerability scanners are a prime example. Such tools and techniques must be evaluated and validated, as these are known as generally having poor precision and recall [12]. To this end, one may use known security vulnerabilities. Nevertheless, to assess the detection methods and scanners in scenarios well beyond the known vulnerabilities, such as newly developed software, one may inject security defects in target systems and validate whether those defects are correctly identified.
- *Software vulnerability seeding.* The technique of defect seeding [19] consists in deliberately inserting a number of bugs in a computer program before verification. Then, once the verification process is complete, one can estimate the number of defects *remaining* in the code through extrapolation based on the number of bugs that were found from those that were deliberately inserted. An equivalent process can be used to estimate the number of

vulnerabilities remaining in a given program and, ultimately, to decide whether the code is ready for deployment or needs to go through additional vulnerability removal processes.
- *Training and evaluating teams and processes.* Security teams and software processes aiming to improve the security of computer systems must be trained and evaluated [21]. In essence, one cares to improve and evaluate the ability to identify security vulnerabilities during software inspections. In this context, being able to emulate representative security bugs is a fundamental step to appraise whether those vulnerabilities are detected. Fagan [10], in the historic paper on design and code inspections, highlights the importance of the feed-forward step to increase defect detection efficiency, by improving the knowledge of which defect types to look for and the ways in which to find each defect type.

The field study presented in this paper consists of analysing a significant number of vulnerabilities in three well-known projects written in the C programming language. Each entry in the CVE database contains a patch file with the exact modifications carried out to correct the vulnerability. Comparing those modifications to the software fault injection operators identified by Durães *et al.* [9], and thereby classifying the modifications according to the Orthogonal Classification Method, we were able to identify a few important patterns that make it feasible to inject vulnerabilities at the source-code level. Hence, the main contributions of this paper are the following:

1. A precise characterization of the nature of software vulnerabilities, conducted by analyzing a large set of real vulnerabilities publicly known for the Linux kernel, the Xen hypervisor, and the OpenSSH tool. This field study contributes to understanding which software defects become security vulnerabilities, in the C programming language that is widely used for developing systems software.
2. An analysis of the typical emulation operators needed to insert vulnerabilities in C programs and the typical distribution (whether it is across a single function, multiple functions, and multiple files). An important observation from the results is that the majority of vulnerabilities affect a single function. Some emulation operators are deterministic in the sense that an injection is functionally equivalent to an actual vulnerability, and we propose to use non-deterministic emulation for some of the remaining operators (for which deterministic emulation is unfeasible).
3. A practical approach to emulate security vulnerabilities at the source-code level, applying a software fault injector with the necessary emulation operators and their distribution to emulate representative security bugs.

The remainder of this paper is organized in the following manner. Section 2 presents the background in the area of software faults and security vulnerabilities. Section 3 details the process used to obtain and analyze the real vulnerabilities. Section 4 presents the field study and the main observations derived from the results. Section 5 proposes a practical approach to emulate software vulnerabilities. Section 6 concludes and summarizes the paper.

2 Background

There have been a number of studies on the nature of software faults specifically aiming at classifying them systematically by examining software patches and defect corrections. Chillarege *et al.* [5] and Christmansson *et al.* [6] were among the early studies leading to the Orthogonal Defect Classification method and related approaches developed to improve software quality. Such studies have recognized the importance of field data as a means to improve the knowledge of how software failures manifest in real systems [8] and to increase defect detection efficiency [10].

Maxion *et al.* [17] have investigated a few specific defect types introduced by programmers writing software in the C programming language. This research has subsequently led to more general approaches that systematically classify the most frequently occurring software defects [9] and that allow practical fault injection at the source code level [20]. These and other related efforts have resulted in a specific subset of emulation operators [7] regarded as representative for the purpose of software fault injection. Unlike these related research efforts, the present paper focuses specifically on security vulnerabilities rather than software defects in general.

In a previous publication, we have examined whether the emulation operators used for software fault injection could be used to emulate software vulnerabilities [3]. That article shows that only a minority of vulnerabilities can be emulated accurately with the fault model for software faults. In comparison with that work, the field data analyzed in the present study is supported by a much larger set of security vulnerabilities, aiming to characterize the typical programming mistakes that compose security vulnerabilities. Furthermore, unlike this previous work, the present study proposes a practical approach to emulate representative vulnerabilities by combining multiple emulation operators and includes a novel approach for non-deterministic emulation of some operators for which exact emulation is unfeasible.

This paper builds upon previous research to investigate whether the principles of software fault injection allow emulation of security vulnerabilities. A conceptually similar study [11] has been conducted for software written in the PHP programming language, targetting web applications. The authors subsequently proposed an approach to validate vulnerability scanners for web applications [12] and an approach to perform vulnerability and attack injection [13]. Unlike these related publications, the present paper addresses the context of systems software written in the C language. In this context, the results and observations are significantly different. Namely, the single most frequent defect in PHP programs was found to be a missing function call, whereas the present study for the C language displays a much more rich and diverse set of common defects arising in real vulnerabilities and exposes the need to combine multiple fault emulation operators.

Software fault injection consists in emulating common programming errors by modifying the source code (or the binary code) according to certain rules [14]. One may use this method with several objectives, such as to validate and guide

the design of fault tolerance mechanisms or to evaluate the behavior and failure modes of applications under activated software faults. To achieve this, it is particularly relevant to ensure the representativeness of the faults injected, *i.e.*, to ensure that the injected faults reflect likely defects in the software which remain after software verification (functional testing, design inspections, code reviews, and so on). Although one may not know in advance which faults will arise in a software project, fault representativeness may be achieved by adopting a fault model that uses knowledge from previous defects found in real software projects. One may regard the emulation of security vulnerabilities as an instantiation of software fault injection with security-specific fault operators and combinations of fault operators.

Performing software fault injection comes with the unavoidable complexity concerning the large number of faults to be injected and the time needed to inject them. To address this problem one may apply different techniques to reduce the temporal effort without sacrificing the representativeness and the validity of the results. An improvement consists in sorting and instantiating each software fault according to the McCabe complexity number [18], also known as cyclomatic complexity, of the function where it is applied.

3 Method

In order to characterize the key elements of a representative security vulnerability, this paper relies on the analysis of 147 publicly known vulnerabilities of the Linux kernel, the Xen hypervisor, and the OpenSSH tool. These software projects are written in the widely used C programming language, have a large installed base, and the correction of known vulnerabilities can be found as patch files released in public databases and repositories.

We adopt the definition used in Orthogonal Defect Classification (ODC), according to which a *software defect* is *"a necessary change to the software"* [4]. Then, in the context of our analysis, a *software vulnerability* is defined as a software defect that can be exploited to compromise a system's information security, thereby having an impact on the security requirements of confidentiality, integrity, and availability [21]. Hence, software vulnerabilities are a subset of all software defects.

A software vulnerability can be formally specified by means of one or more patch files containing operations of addition, removal, or modification of source code lines. This captures the notion that defects are composed of *missing*, *extraneous*, and *wrong* constructs, which are in line with ODC's distinction between something that is missing and something that is incorrect. Our goal is to map each operation required to fix a vulnerability onto the *operators* needed to reverse that fix. In other words, we are able to reintroduce a certain vulnerability, in a later version of the same software, by applying a specific combination of emulation operators. Therefore, each operator aims to emulate a unitary programming mistake (in the present case, a mistake related to a software vulnerability). We consider ODC's defect types listed below.

- Algorithm – Programming errors that affect correctness or efficiency of some computation and that can be corrected through reimplementation of an algorithm without requiring changes to the architecture or design.
- Assignment – Defects that affect localized parts of the source code and consist of missing or incorrect variable assignments or initializations.
- Checking – Incorrect or missing program constructs where selection or loop conditions are affected and therefore influence program logic.
- Function – Defects that affect a significant part of a program, including for instance functionality that is missing or large algorithms, and that would require changes to the design or architecture of the program.
- Interface – Programming errors that involve interaction among functions or components through incorrect parameters or incorrect return values.

By adopting the Orthogonal Defect Classification method we are implicitly following the defect types specified by Chillarege *et al.* [5]. To ensure that defect types are orthogonal, ODC stipulates that a programmer implicitly chooses the defect type by making a correction. Consequently, when a correction is translated into a software patch, one may examine the exact changes that were made to the source code to determine the defect type.

We selected three open-source software projects based on their large user base, public availability of CVE repositories, and coding language (our study is limited to the C programming language). The goal is to base our analysis upon a representative set of vulnerabilities. The Linux kernel [15] is used in a wide range of systems and supports numerous hardware architectures and devices. A vulnerability in the Linux kernel can be exploited to gain unauthorized access to a machine, often including administration access. The Xen hypervisor [2] is extensively used in cloud computing datacenters by major cloud providers. A vulnerability in Xen may allow an attacker to interfere with, or gain control over, virtual machines running on the platform. The OpenSSH tool [16] is a suite of utilities designed to secure network traffic using the SSH protocol. A vulnerability in OpenSSH can allow an attacker to remotely obtain access and execute unauthorized code in a system. All of these projects

We specified and followed a set of rules and inclusion criteria to ensure repeatable analysis of vulnerabilities. These rules stipulate how to obtain vulnerabilities for analysis, how to verify if they are applicable to the analyzed software versions, and how to map the vulnerabilities to emulation operators and classify them. The software vulnerabilities and the respective patch files were collected from public repositories, giving preference to the most official source for this information. In the case of the Xen hypervisor, we used the project's Xen Security Advisories (XSA) official repository. For the Linux kernel and the OpenSSH tool we used the CVEDetails repository. Our analysis covers recent vulnerabilities applicable to the selected software versions.

Each vulnerability included in the field study obeyed several criteria. First, we considered vulnerabilities affecting the software projects and versions selected for the study sequentially by date of publication. We collected the patch files from the vulnerability databases, starting with the most recent toward the

oldest vulnerability, and performed the analysis without modifying neither the patch file nor the project code. Then, we manually analyzed each patch file to confirm that only C code was being altered and that source code macros were not part of the changes. Finally, only those vulnerabilities directly affecting the selected software projects were included, while we excluded those affecting other projects that may be packaged or that are somehow related (*e.g.*, the *qemu* project is packaged by the Xen project) or components not belonging to the project's core (*e.g.*, the *libxl* utility is a part of Xen but does not belong to the core of the project, since Xen does not require it to execute). This resulted in 147 vulnerabilities in total, corresponding to (at the time of the study) all the vulnerabilities that could be obtained for the OpenSSH tool that obeyed the criteria for inclusion and the majority of vulnerabilities for the other two projects.

After collecting the vulnerabilities, each patch file was analyzed to map the actual code changes to fault emulation operators. This process was manually conducted by the authors, by comparing the faults classified according to ODC with the individual operations needed to reproduce each vulnerability (*i.e.*, the changes required to reintroduce a vulnerability given the source code that corrected it). We aimed for *functional equivalence* when applying changes, *i.e.*, the resulting code after applying the operators should be functionally the same as the real vulnerabilities. In some cases there may be more than one way of emulating a given vulnerability and for those cases we selected the alternative which minimizes the total number of emulation operators applied.

The example provided below aims to clarify the mapping process in detail. Considering that we analyze a patch file including the information shown in Figure 1, one may observe that this vulnerability was fixed by adding a new function call (with return value) and a new *if* construct. These additions are marked with the plus sign.

```
1     if( nestedhvm_vcpu_in_guestmode(v) )
2         nvmx_idtv_handling();
3   + mode = vmx_guest_x86_mode(v);
4   + if(mode == 8 ? !is_canonical_address(regs->rip) : regs->rip!=regs->_eip){
5   +   struct segment_register ss;
6   +   gprintk(XENLOG_WARNING, "Bad␣rIP␣%lx␣for␣mode␣%u\n", regs->rip, mode);
7   +   vmx_get_segment_register(v, x86_seg_ss, &ss);
8   +   if ( ss.attr.fields.dpl ) {
9   +     __vmread(VM_ENTRY_INTR_INFO, &intr_info);
10  +     if ( !(intr_info & INTR_INFO_VALID_MASK) )
11  +       hvm_inject_hw_exception(TRAP_gp_fault, 0);
12  +     if ( mode == 8 )
13  +       regs->rip = (long)(regs->rip << (64-VADDR_BITS)) >> (64-VADDR_BITS);
14  +     else
15  +       regs->rip = regs->_eip;
16  +   }
17  +   else
18  +       domain_crash(v->domain);
19  + }
20   }
```

Fig. 1: Excerpt from the patch file associated with CVE-2016-2271 (XSA-170).

Hence, to emulate this vulnerability, we need to remove that same function call and the *if* construct. This removal operation can be mapped to one instance of the MFC operator and one instance of the MIFS operator (see Table 7). Using this information, we run our fault injection tool on the source code of the project with the patch shown in Figure 1 applied (*i.e.*, the vulnerability is fixed in the source code) and verify whether our tool has applied the MFC operator (to line 3) and, separately, the MIFS operator to the *if* construct including statements (lines 4 through 19). If it did, then the software fault operators are capable of emulating that vulnerability.

4 Field study

We collected and analyzed 147 real vulnerabilities in order to characterize the typical programming mistakes that cause security defects. Each entry in a CVE repository provides patch files (often a single file) that correct a specific vulnerability. Those patch files were the object of our analysis and the resulting dataset, along with the source files, shall be publicly released with the paper. The remainder of this section presents the results obtained in the field study.

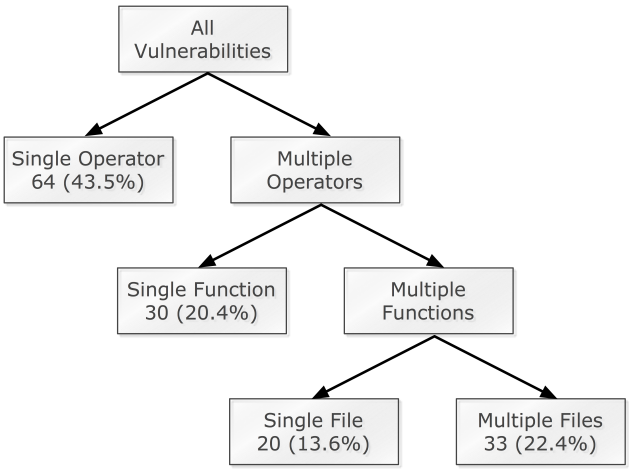

Fig. 2: Distribution of emulation operators across multiple functions and multiple files.

Each vulnerability was mapped onto a set of emulation operators necessary to inject the vulnerability, as described in the preceding section. Figure 2 shows that 64 out of 147 vulnerabilities (43.5%) consist of a unitary programming mistake (*i.e.*, a single instance of an operator). Furthermore, in 30 cases (20.4%) the vulnerability affected a single function but was composed of multiple operator instances (*i.e.*, multiple programming mistakes in the same function). In the remaining cases, 20 vulnerabilities involved multiple functions in a single file and 33 vulnerabilities affected multiple files.

Thus, it is a relevant observation that the majority of vulnerabilities affect a single file, either requiring a single operator (43.5%) or multiple operators (20.4%). These add up to 63.9% of all cases – nearly two thirds of the vulnerabilities examined. This is an important observation, because emulating vulnerabilities involving a single function is less complex than doing so for multiple functions (and far less complex than those involving multiple files).

Figure 3 shows how many instances of emulation operators are needed for each vulnerability. The histogram shows the absolute number of vulnerabilities which consist of 1 operator instance, 2 operator instances, and so on. Note that in case a given emulation operator appears more than once in a single vulnerability, all those instances are counted in the histogram.

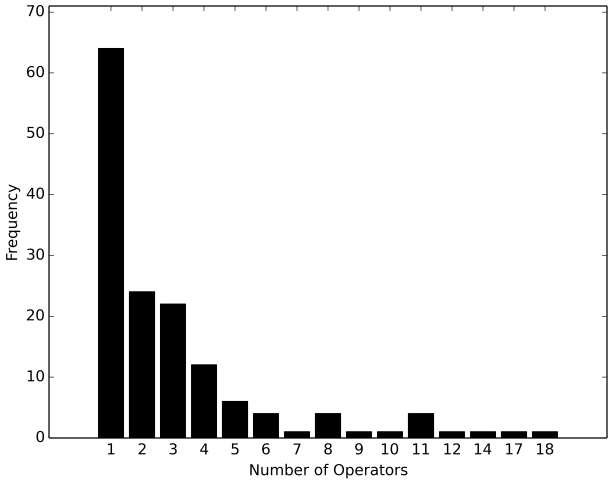

Fig. 3: Absolute frequency of vulnerabilities over the number of operator instances necessary to emulate them.

One can observe in Figure 3 that the vast majority of vulnerabilities consist of up to three programming mistakes (*i.e.*, three instances of emulation operators). A single operator is sufficient in 43.5% of the vulnerabilities, two operators are needed in 16.3% of the vulnerabilities, and three operators are needed in 15.0% of the vulnerabilities. Thus, it is a relevant observation that 74.8% – nearly three fourths of the vulnerabilities are composed of up to three programming mistakes.

Due to the combinatorial nature of the problem, emulating vulnerabilities with 3 operators is much less complex than emulating vulnerabilities with, for example, 5 operators. Nevertheless, if we wish to increase the coverage of our analysis, 5 operators cover up to 89.1% of all vulnerabilities. Table 1 shows

the same information as the histogram, broken down by software project, from which similar observations can be drawn.

Table 1: Absolute number of vulnerabilities over the number of operator instances for each software project.

	Number of Operators						
Software	1	2	3	4	5	6	≥ 7
Xen	18	8	7	7	4	0	6
Linux Kernel	25	9	5	4	0	2	5
Open SSH	21	7	10	1	2	2	4
Total	64	24	22	12	6	4	15

Having presented the overall results regarding the number of operators and their distribution across multiple functions and multiple files, Table 2 shows a detailed analysis of the actual programming mistakes found in all software projects, specifically for algorithm defects. The table shows the number of vulnerabilities in which each operator was found. If an operator appears more than once in the same vulnerability it is nevertheless counted as a single vulnerability in which it appears.

Table 2: Detailed description and frequency of algorithm faults.

Fault nature	Fault specific types	Xen	Linux Kernel	Open SSH	Total
Missing construct	Missing function call (MFC)	13	10	9	32
	Missing if construct plus statements (MIFS)	16	23	18	57
	Missing if-else construct plus statements (MIES)			1	1
	Missing if construct plus statements plus else before statements (MIEB)	2	1		3
	Missing if construct plus else plus statements around statements (MIEA)		2	1	3
	Missing case: statement(s) inside a switch construct (MCS)	1		2	3
	Missing branch construct - continue (MBCC)	1			1
	Missing small and localized part of the algorithm (MLPA)		1		1
	Missing ELSE construct plus statements (MES)	3			3
Wrong construct	Wrong function called with same parameters (WFCS)	1	2	1	4
	Wrong function called with different parameters (WFCD)	4	1	2	7
	Wrong algorithm - small sparse modifications (WALD)	1	2	1	4
	Wrong algorithm - code was misplaced (WALR)	1	5	3	9
	Wrong variable used in switch construct (WVS)	1			1
Extraneous construct	Extraneous function call (EFC)	8	6	2	16
	Extraneous else statement (EES)	1			1
	Extraneous IF construct around statements (EIA)	2	1		3
	Extraneous IF construct plus statements (EIFS)	3	4	3	10
	Extraneous break in case (EBC)	1			1
	Extraneous case statement (ECS)	2			2
Total faults found		61	58	61	162

One can observe in Table 2 that algorithm faults predominantly consist of MFC, MIFS, WALR, EFC, and EIFS types (for example, missing function call (MFC) was found in 32 vulnerabilities, and missing *if* construct plus statements (MIFS) was found in 57 vulnerabilities). The remaining algorithm defects are relatively infrequent, as those appear in 7 or less vulnerabilities (*i.e.*, less than 5% of the vulnerabilities).

Assignment faults are less common than algorithm faults, as shown in Table 3. The table shows that assignment faults are very diverse, and only the MVAV operator (missing variable assignment using a value) appears in more than 5% of the vulnerabilities.

Table 3: Detailed description and frequency of assignment faults.

Fault nature	Fault specific types	Xen	Linux Kernel	Open SSH	Total
Missing construct	Missing variable initialization using a value (MVIV)	1	2	4	7
	Missing variable initialization using an expression (MVIE)	2	3	1	6
	Missing variable assignment using a value (MVAV)	6	2	1	9
	Missing variable assignment using an expression (MVAE)	3	1	1	5
	Missing variable auto-decrement (MVAD)		1		1
Wrong construct	Wrong logical expression used in assignment (WVAL)	1			1
	Wrong arithmetic expression used in assignment (WVAE)	3			3
	Wrong value used in variable initialization (WVIV)		1	1	2
	Wrong value assigned to variable (WVAV)	1	2	2	5
	Wrong data types or conversion used (WSUT)	2	2	2	6
Extraneous construct	Extraneous variable assignment using a value (EVAL)	1		1	2
	Extraneous variable assignment using another variable (EVAV)	2		2	4
	Extraneous variable assignment with an expression (EVAE)	2			2
Total faults found		24	14	15	53

Table 4 shows that three operators dominate checking faults, namely MLOC, MLAC, and WLEC. The MLAC and MLOC operators are fairly simple to emulate by removing parts of expressions used in branch conditions (IF statements in nearly all cases). The WLEC operator consists of an incorrect branch condition and is therefore difficult to emulate exactly. For cases such as this one, we propose non-deterministic emulation in the implementation section of the paper.

Function faults are detailed in Table 5. Such faults usually require large modifications to functionality, but are also infrequent (only 4 cases out of 147 vulnerabilities).

The detailed description of interface faults is presented in Table 6. One can observe that interface faults, as a class, appear in 32 out of 147 vulnerabilities. Nevertheless, all defect types occur in less than 5% of all vulnerabilities.

The results of this field study are in line with previous research conducted on programs written in the C language, in the sense that algorithm, assignment, and checking faults are the most frequent. However, upon detailed analysis, the

Table 4: Detailed description and frequency of checking faults.

Fault nature	Fault specific types	Xen	Linux Kernel	Open SSH	Total
Missing construct	Missing if construct around statements (MIA)	1	1	4	6
	Missing "OR EXPR" in expression used as branch condition (MLOC)	4	4	2	10
	Missing "AND EXPR" in expression used as branch condition (MLAC)	3	3	4	10
Wrong construct	Wrong logical expression used as branch condition (WLEC)	7	4	1	12
	Wrong arithmetic expression in branch condition (WAEC)		1		1
Extraneous construct	Extraneous "OR EXPR" in expression used as branch condition (ELOC)	1			1
Total faults found		16	13	11	40

Table 5: Detailed description and frequency of function faults.

Fault nature	Fault specific types	Xen	Linux Kernel	Open SSH	Total
Wrong construct	Wrong algorithm - large modifications (WALL)	1		3	4
Total faults found		1		3	4

Table 6: Detailed description and frequency of interface faults.

Fault nature	Fault specific types	Xen	Linux Kernel	Open SSH	Total
Missing construct	Missing return statement (MRS)	1	1	2	4
	Missing parameter in funtion call (MPFC)	2		3	5
Wrong construct	Wrong logical expression in param. of func. call (WLEP)	1		1	2
	Wrong arithmetic expression in param. of func. call (WAEP)	1			1
	Wrong variable used in parameter of function call (WPFV)	2		1	3
	Wrong value used in parameter of funtion call (WPFL)	1	3	3	7
	Miss by one value in parameter of function call (WPFML)	1			1
	Wrong return value (WRV)	1	4	2	7
Extraneous construct	Extraneous AND EXPR in branch condition (ELAC)	1		1	2
Total faults found		11	8	13	32

most frequent defect types differ to some extent when compared to those found in previous studies on software defects in general [6, 9]. The most frequent operators, shown in Table 7, consist of nine common programming mistakes that appear (each one) in more than 5% of all vulnerabilities.

The results summarized in Table 7 show that new operators, which have not been considered for the emulation of software faults, are needed to emulate software vulnerabilities. Specifically, the WLEC, EIFS, EFC and WALR types of defect are relatively frequent in software vulnerabilities, whereas these were

Table 7: Most frequent fault types occurring in software vulnerabilities.

Operator	Description	Type	#Vulnerabilities
MIFS	Missing IF construct plus statements	Algorithm	57 (38.8%)
MFC	Missing function call	Algorithm	32 (21.8%)
EFC	Extraneous function call	Algorithm	16 (10.9%)
WLEC	Wrong logical expression used as branch condition	Checking	12 (8.2%)
EIFS	Extraneous IF construct plus statements	Algorithm	10 (6.8%)
MLOC	Missing OR EXPR in expression used as branch condition	Checking	10 (6.8%)
MLAC	Missing AND EXPR in expression used as branch condition	Checking	10 (6.8%)
WALR	Wrong algorithm - code was misplaced	Algorithm	9 (6.1%)
MVAV	Missing variable assignment using a value	Assignment	9 (6.1%)

not previously included among the most representative types in software fault injection.

5 Practical emulation of software vulnerabilities

Having extensively studied the characteristics of software vulnerabilities, by analyzing the three software projects included in our field study, we now describe a practical approach for emulating vulnerabilities. While software fault injection has not been designed to emulate software vulnerabilities, it already provides the means to emulate the simplest ones and the techniques behind it are applicable with some modifications for our purpose. In this section we describe the adaptation of a software fault injection tool to emulate software vulnerabilities.

5.1 Overview

To emulate security vulnerabilities one must modify programs according to a set of operators that specify the modification that should take place, either at the source code or at the binary level. Although some adaptations are necessary, this process is conceptually similar to the emulation of software faults in general. Therefore, as the basis we use a software fault injector, which specifies a subset of the fault emulation operators, along with the associated constraints, which stipulate the conditions under which an operator may be applied. The injection flow remains largely unchanged, with the exception that a mechanism is needed to combine multiple operator instances to produce a single security vulnerability.

It should be highlighted that this approach to emulate software vulnerabilities is complete, in the sense that all representative security vulnerabilities can be injected. In other words, taking the fault model derived from the field study

as a set of representative vulnerabilities, one may emulate all such vulnerabilities (within the restrictions of time and limitations of the parsing stage in which the abstract syntax tree is built). However, the converse is untrue as an injected fault may or may not be a true security vulnerability. Consequently, when this approach is used to evaluate a vulnerability detection method, scanner, or team process, one should not aim for full detection as some injected faults may not be true vulnerabilities.

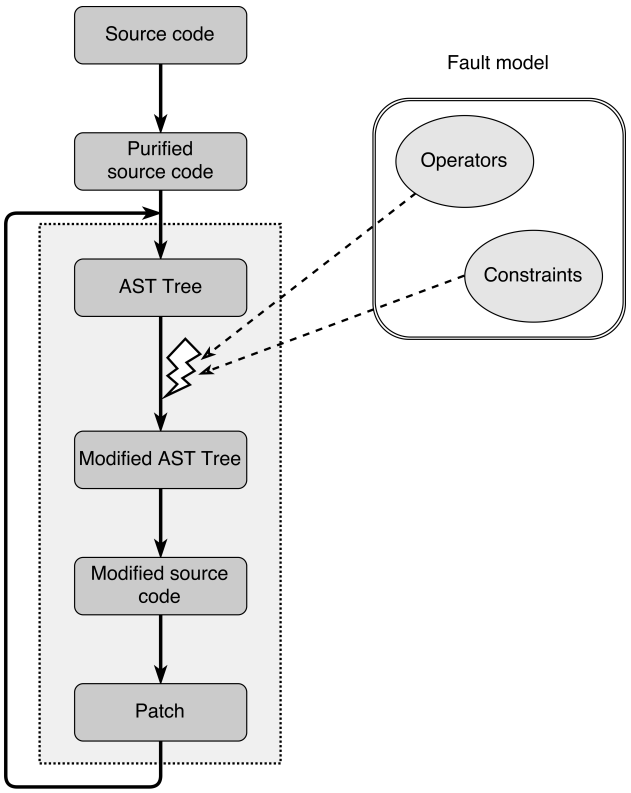

Fig. 4: Flow for software fault injection at the source code level.

Our vulnerability injector has been developed as an extension to the software fault injection tool in the ucXception suite [20]. This tool allows software faults to be injected at the source code according to the shortlist of fault operators [9] for programs in the C language. Our tool starts by parsing the source code and building an abstract syntax tree (AST) using the Eclipse CDT library. This process works generally well but is limited by some restrictions of the library in handling macros and C preprocessor directives. Once the AST is constructed, as shown in Figure 4, the various fault operators consist of Java functions that

operate directly over the syntax tree to introduce the intended fault if and only if all constraints are met.

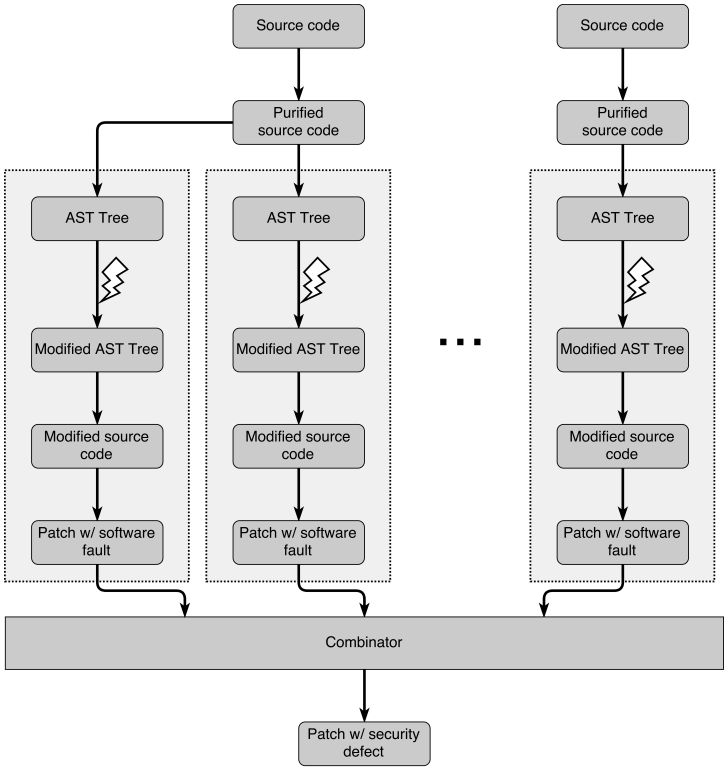

Fig. 5: Combined flow for emulating security vulnerabilities.

The flow of software fault injection campaigns in the ucXception tool is depicted in Figure 4. The tool produces a set of patches – each consisting of a single emulation operator instance. Figure 5 conveys the flow to emulate software vulnerabilities. A software vulnerability usually requires multiple instances of emulation operators to be combined (as observed in the field study, the fault model consists of multiple operators applied to a single function at a time). This key difference requires the tool to join all the source code modifications into a single patch file that represents one software vulnerability.

Returning to the example provided in Figure 1, the tool produces one patch for each emulation operator instance, including in that example one MFC and one MIFS operator (among many others). This is the case given, as input, the source code of the software with the vulnerability fixed, but could also be some other source code containing potential injection locations. Those two patch files are applied to the original software to obtain a combined insertion of two operators, thereby emulating the security vulnerability shown in the example.

As a result of our field study, we conclude that the most commonly used fault model for emulating software faults [9, 20], while valid in its context, is not a representative fault model for emulating security vulnerabilities. The usage of a fault model tailored for the emulation of software vulnerabilities, which shares similarities but is adapted for this specific purpose, yields better coverage (*i.e.*, is able to emulate a greater number of vulnerabilities) and is more efficient, as it forgoes the individual operators that have been shown not to be required for emulating software vulnerabilities. In Section 4 we have described the fault model and its operators, of which some were already included in the original model of software faults, while others had not been considered prior to the present work. Having described the modifications to the injection flow, we now turn to the details of the new fault operators and the changes needed to the ones extended from previous research, all of which are needed to cover the list shown in Table 7 with the objective of emulating software vulnerabilities.

5.2 Relaxation of constraints

Most of the operators in the fault model for emulating software vulnerabilities were already present in the fault model for software faults. Thus, it is possible to take advantage of this fact and reuse their implementation in existing software fault injection tools. However, in some occasions, small changes are required so that they become adequate for emulating vulnerabilities. This is the case with the MFC (Missing function call) operator. In software fault injection this operator has the following constraints [20]:

- C01 – Return value of the function must not be used
- C02 – Call/Assignment/The if construct/The statements must not be the only statement in the block

However when applying this operator for the purpose of emulating software vulnerabilities, constraint C01 must be ignored. Dropping this constraint comes as a result of our field study, given the large number of situations where one was required to remove a function call even though its return value was assigned to a variable.

5.3 Extraneous operators

The model of software faults features solely operators that imply modification (those that start with a 'W') or removal (those that start with an 'M'), which are, by nature, moderately simple to implement. The proposed fault model of software vulnerabilities sees the appearance of 'extraneous operators' (*i.e.*, operators that imply adding new constructs to the source code). These operators present a new challenge: choosing what to add and where to add the new constructs.

In particular, if we look at the EFC (Extraneous function call) operator, it is easily understood that the fault injector must determine to which function

it should add a call. In this scenario, the set of possible functions includes not only the functions defined in the current file, as well as the functions defined in the project and which are visible from that file and also the functions of all included libraries. Not only that, but each function has its own parameters which must receive a value. Thus, for this particular operator, there is also the challenge of deciding which values or variables should be passed to the function. If we look at the EIFS (Extraneous IF construct plus statements) operator we see a similar scenario. Not only do we need to choose which condition will be used (as discussed in the next subsection) but also which statements shall be inside the If block.

In all extraneous operators there is one more decision that must be taken: where to apply the operator, or, in other words, where to add a new construct. Whereas non-extraneous operators are regulated by constraints that strongly limit the locations in the code where they can be applied, for example, the MVAV (Missing variable assignment using a value) operator can only be applied if an assignment is not inside of a for loop [9], extraneous operators can be applied almost anywhere in the code (*e.g.*, inside of every function) and are less restricted by constraints.

The outcome of these characteristics that are inherent to extraneous operators is not only the increased difficulty when building the fault injection tool, but also the very high number of possible combinations that can be produced. This means that extraneous operators contribute more significantly to the combinatorial cost of performing fault injection than other operators. This is particularly important to the emulation of software vulnerabilities since one vulnerability may imply applying operators more than once.

5.4 Approximate emulation

Two of the newly introduced operators present a challenge that has not yet been addressed by previous research in software fault injection or related areas. The EIFS (Extraneous IF construct plus statements) and WLEC (Wrong logical expression used as branch condition) operators imply the creation of a new logical expression. In the case of EIFS, a new logical condition has to be created from the ground up and will determine whether the flow of execution should enter inside of the statements enclosed by the if, while in WLEC, the original logical expression has to be changed as to become wrong but syntactically and programmatically correct. Before a careful analysis of this problem is performed, the domain of possible conditions that can be used appears tremendously large and far from feasible to emulate in practice. However, this problem can be simplified without loss of fidelity, as described in the next paragraphs.

We propose the concept of *approximate emulation* of operators. Approximate emulation is a novel idea that uses non-determinism to solve the aforementioned problem of creating logical expressions. The key insight behind approximate emulation is that a logical expression always yields either *true* or *false*. Thus by replacing a logical expression with a function that randomly (and hence

non-deterministically) returns *true* or *false*, we are able to emulate the outcome of any logical expression. The disadvantage of this approach is that the user must be aware of this peculiarity and take into account that the execution of the program (that has an operator that requires approximate emulation) will have a non-deterministic behavior (*i.e.*, executing the same program multiple times can lead to multiple different outcomes depending on the random choices).

```
while (<random logical condition>) {
    (...)
}
```

Fig. 6: Example of a situation that leads to infinite possible paths.

To construct random logical conditions one may use `rand() % 2`, for instance, as the expression to compute a pseudo-random logical value *true* or *false*. The `rand()` function is provided by `stdlib.h` along with the `srand()` function that may be used to set the initial seed, thereby selecting a specific sequence of random numbers. By executing, for example, `srand(time(NULL))` during initialization, one may obtain pseudo-random executions in which the defect may or may not manifest.

```
for (int i = 0; i < some_variable; i++) {
    if (<random logical condition>) {
        (...)
    }
}
```

Fig. 7: Example of a condition where the outcome may vary in every execution.

It is worthy to highlight that most vulnerabilities examined in the field study are deterministic, *i.e.*, wrong conditions always compute the same incorrect value. Nevertheless, there are exceptions such as missing initializations and code involving I/O that result in implicit non-determinism. In our context, explicitly using non-determinism to perform approximate emulation resembles Monte Carlo approaches and, consequently, one may need to execute the same program multiple times to activate an emulated vulnerability.

By applying non-determinism we are able to emulate the EIFS and WLEC operators, which were found to appear frequently in software vulnerabilities although these had not been considered as relevant by previous research. Situations such as the one depicted in Figure 6 lead to an infinite number of possible executions and situations such as the one in Figure 7, where the result of the condition can vary at each execution of the main loop, cannot be deterministically represented (except certain situations where loop unrolling [1] can be applied).

Finally, we argue that a non-deterministic vulnerability is still a vulnerability as it may be exploited with some probability. As a result, we are able to emulate all the fault operators found to be representative of software vulnerabilities and the combined flow provides the means to aggregate multiple fault operators into a single emulated vulnerability.

6 Conclusion

This paper presents a field study to characterize the most frequent programming mistakes that cause security vulnerabilities and a practical approach to emulate those vulnerabilities. To this end, we analyzed 147 publicly known vulnerabilities of the Linux kernel, the Xen hypervisor, and the OpenSSH tool, to determine what composes a representative security vulnerability in the C programming language. We conclude that a typical vulnerability affects a single function, as this was observed in two thirds of the cases. Moreover, a typical vulnerability consists of no more than 3 fault operator instances, as this was observed in three fourths of the cases.

Furthermore, we find that nine specific emulation operators appear, each one, in more than 5% of the vulnerabilities. These are the most frequent programming mistakes found in our field study. Compared to previous studies on software faults in general, we conclude that four new emulation operators must be added for adequate emulation of vulnerabilities. Moreover, unlike software faults in general, emulating vulnerabilities requires combining multiple operator instances rather than applying one emulation operator at a time.

A practical approach for emulating software vulnerabilities is presented and we find that existing software fault injection techniques are suitable to emulate software vulnerabilities. However, the set of emulation operators (or mutations) should consist of the specific types found to be representative of software vulnerabilities and the injection procedure should combine multiple emulation operators in each injection. This, in turn, increases the complexity and the number of combinations when compared to standard software fault injection. Moreover, this paper proposes non-deterministic emulation for some of the new operators for which exact emulation would be too complex. The expected impact of this study is to allow practical emulation of software vulnerabilities for software quality and security assessment.

As future work, we envisage the model of vulnerabilities and the injection method to be applied to validate vulnerability scanners and static analysis tools, to assess vulnerability detection methods, to enable software vulnerability seeding, and to train and evaluate teams and processes regarding the efficiency of defect detection.

References

1. Aho AV, Lam MS, Sethi R, Ullman JD (2007) Compilers: Principles, Techniques, and Tools, 2nd edn. Pearson/Addison-Wesley
2. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A (2003) Xen and the art of virtualization. SIGOPS Oper Syst Rev 37(5):164–177, DOI 10.1145/1165389.945462
3. Cerveira F, Barbosa R, Mercier M, Madeira H (2017) On the emulation of vulnerabilities through software fault injection. In: 2017 13th European Dependable Computing Conference (EDCC)

4. Chillarege R (1996) Orthogonal defect classification. In: Lyu MR (ed) Handbook of Software Reliability Engineering, IEEE CS Press and McGraw-Hill, chap 9, pp 359–400
5. Chillarege R, Bhandari IS, Chaar JK, Halliday MJ, Moebus DS, Ray BK, Wong MY (1992) Orthogonal defect classification—A concept for in-process measurements. IEEE Transactions on Software Engineering 18(11):943–956
6. Christmansson J, Chillarege R (1996) Generation of an error set that emulates software faults based on field data. In: Proceedings of the Twenty-Sixth International Symposium on Fault-Tolerant Computing, IEEE, Washington, pp 304–313
7. Cotroneo D, Natella R (2013) Fault injection for software certification. IEEE Security & Privacy 11(4):38–45, DOI http://dx.doi.org/10.1109/MSP.2013.54
8. Cotroneo D, Pietrantuono R, Russo S, Trivedi KS (2016) How do bugs surface? a comprehensive study on the characteristics of software bugs manifestation. Journal of Systems and Software 113:27–43
9. Duraes JA, Madeira HS (2006) Emulation of software faults: A field data study and a practical approach. IEEE Transactions on Software Engineering 32(11):849–867, DOI 10.1109/TSE.2006.113
10. Fagan ME (1976) Design and code inspections to reduce errors in program development. IBM Systems Journal 15(3):182–211
11. Fonseca J, Vieira M (2008) Mapping software faults with web security vulnerabilities. In: 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN), pp 257–266, DOI 10.1109/DSN.2008.4630094
12. Fonseca J, Vieira M, Madeira H (2007) Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks. In: 13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007), pp 365–372, DOI 10.1109/PRDC.2007.55
13. Fonseca J, Vieira M, Madeira H (2009) Vulnerability & attack injection for web applications. In: 2009 IEEE/IFIP International Conference on Dependable Systems Networks, pp 93–102, DOI 10.1109/DSN.2009.5270349
14. Hsueh MC, Tsai TK, Iyer RK (1997) Fault injection techniques and tools. IEEE Computer 30(4):75–82, URL http://doi.ieeecomputersociety.org/10.1109/2.585157
15. Love R (2005) Linux Kernel Development (2Nd Edition) (Novell Press). Novell Press
16. Lucas MW (2012) SSH Mastery: OpenSSH, PuTTY, Tunnels and Keys. Tilted Windmill Press
17. Maxion RA, Olszewski RT (2000) Eliminating exception handling errors with dependability cases: A comparative, empirical study. IEEE Trans Software Eng 26(9):888–906, URL http://dx.doi.org/10.1109/32.877848; http://doi.ieeecomputersociety.org/10.1109/32.877848
18. McCabe TJ (1976) A complexity measure. IEEE Transactions on Software Engineering SE-2(4):308–320, DOI 10.1109/TSE.1976.233837

19. McConnell S (1997) Best practices: Gauging software readiness with defect tracking. IEEE Software 14(3):136, 135
20. Pereira G, Barbosa R, Madeira H (2016) Practical emulation of software defects in source code. In: 2016 12th European Dependable Computing Conference (EDCC), pp 130–140, DOI 10.1109/EDCC.2016.19
21. Stallings W, Brown L (2011) Computer Security: Principles and Practice, 2nd edn. Prentice-Hall, Inc.