Experience Report: Security Vulnerability Profiles of Mission Critical Software: Empirical Analysis of Security Related Bug Reports

Katerina Goseva-Popstojanova and Jacob Tyo
Lane Department of Computer Science and Electrical Engineering
West Virginia University, Morgantown, WV, USA
Email: Katerina.Goseva@mail.wvu.edu

***Abstract*—While some prior research work exists on characteristics of software faults (i.e., bugs) and failures, very little work has been published on analysis of software applications vulnerabilities. This paper aims to contribute towards filling that gap by presenting an empirical investigation of application vulnerabilities. The results are based on data extracted from issue tracking systems of two NASA missions. These data were organized in three datasets: Ground mission IV&V issues, Flight mission IV&V issues, and Flight mission Developers issues. In each dataset, we identified the security related software bugs and classified them in specific vulnerability classes. Then, we created the vulnerability profiles, i.e., determined where and when the security vulnerabilities were introduced and what were the dominant vulnerabilities classes. Our main findings include: (1) In IV&V issues datasets the majority of vulnerabilities were code related and were introduced in the Implementation phase. (2) For all datasets, close to 90% of the vulnerabilities were located in two to four subsystems. (3) Out of 21 primary vulnerability classes, five dominated: Exception Management, Memory Access, Other, Risky Values, and Unused Entities. Together, they contributed from around 80% to 90% of vulnerabilities in each dataset.**

***Keywords*-application security; software vulnerability; security vulnerability profile; mission critical software.**

I. Introduction

Nowadays space missions provide valuable services to the society – from navigation, to earth observation, weather forecasting, and communication. Consequently, space missions are becoming part of the critical infrastructure and are regularly targeted by attackers. In a typical week NASA experiences 29,000 malicious incidents against its systems, 17,500 suspicious e-mails, and 250 unique incidents against its web sites [1]. Furthermore, cybersecurity threats to space missions are expected to continue to grow in the future [2].

The multi-tiered approach to cybersecurity management integrates the IT service security, data and application security, and infrastructure security [1]. This paper is focused on application security, which is an important aspect of the overall cybersecurity. Thus, once an attacker has gained access to an internet-accessible computer, he/she could use the compromised computer to exploit vulnerabilities on mission computers that could significantly disrupt space flight operations and/or steal sensitive data [3]. Therefore, it is becoming an imperative to use software development and assurance practices that account for cybersecurity concerns.

A security vulnerability is defined as a weakness in a system, application, or network that could be subject to exploitation or misuse that would allow an attacker to compromise any aspect of cybersecurity (i.e., confidentiality, integrity, availability, authentication, authorization, and non-repudiation). The work presented in this paper is focused on application vulnerabilities and is based on utilizing the information provided in issue tracking systems. Specifically, our work is focused on analyzing software bug reports and identifying those that are security related. In this context, we use the terms 'vulnerability', 'security related bug', and 'security issue' interchangeably. Note that our study accounts for vulnerabilities that may have been introduced, found, and fixed throughout the software lifecycle.

In this paper, we introduce the term *security vulnerability profile* which is defined as a set of data that describes where and when the security vulnerabilities were introduced and what were the dominant vulnerability classes. We believe that security vulnerability profiles and their underling trends support in-depth understanding of the nature of vulnerabilities and can help developers and Independent Verification and Validation (IV&V) analysts to prevent, detect, and eliminate the vulnerabilities in the most effective ways, at the most effective time.

The results presented in this paper are based on data extracted from issue tracking systems of two NASA missions. These data were organized in three datasets: Ground mission IV&V issues, Flight mission IV&V issues, and Flight mission Developers issues. In each dataset, we identified the security related bugs and classified them in specific vulnerability classes from the Common Weakness and Enumeration (CWE) view CWE-888 [4], [5]. Then, we created the vulnerability profiles and explored the existence of trends. Our main research questions are as follows:

- RQ1: What are the security issues' categories and types?
- RQ2: Where are security issues located?
- RQ3: When are security issues typically introduced and found?
- RQ4: What are the severity levels of security issues?
- RQ5 What are the dominant classes of security issues?
- RQ6 Are the dominant classes of security issues and the other findings consistent across missions and datasets?

The main findings of our work include:

- Code related security issues dominated both the Ground and Flight mission IV&V security issues, with 95% and 92%, respectively. Therefore, enforcing secure coding

2332-6549/17 $31.00 © 2017 IEEE
DOI 10.1109/ISSRE.2017.42

practices and verification and validation focused on coding errors would be cost effective ways to improve missions' security. (Flight mission Developers issues dataset did not contain data in the Issue Category.)
- The location of security issues, as the location of software bugs in general, followed the Pareto principle. Specifically, for all three datasets, close to 90% of security issues were located in two to four subsystems.
- In both the Ground and Flight mission IV&V issues datasets, the majority of security issues (i.e., 91% and 85%, respectively) were introduced in the Implementation phase. In most cases, the phase in which the security issues were found was the same as the phase in which they were introduced. The most security related issues of the Flight mission Developers issues dataset were found during Code Implementation, Build Integration, and Build Verification; the data on the phase in which these issues were introduced were not available for this dataset.
- The severity levels of most security issues, as the majority of all issues, were moderate in all three datasets.
- Out of 21 primary vulnerability classes, five classes dominated: Exception Management, Memory Access, Other, Risky Values, and Unused Entities. Together, these classes contributed from around 80% to 90% of all security issues, in each dataset. This again proves the Pareto principle of uneven distribution of security issues, in this case across vulnerability classes, and supports the fact that addressing these dominant vulnerability classes provides a cost efficient way to improve missions' security.

The rest of the paper is organized as follows. Section II summarizes the related works. Section III describes the classification approach. The two NASA missions and three created datasets are described in section IV. The results for each dataset are presented in section V, followed by the comparison of the results across all three datasets in section VI. The threats to validity are enumerated in section VII and the conclusion is presented in section VIII.

II. Related Work

While prior research work exists on characteristics of software faults (i.e., bugs) and failures, very little work has been published on analysis of software applications vulnerabilities. We first summarize the papers that explored the characteristics of software faults in general, without any focus on security aspects (i.e., vulnerabilities).

Fenton and Ohlsson studied a large telecommunication application and focused on a range of software engineering hypotheses related to the Pareto principle of distribution of faults and failures, the use of early fault data to predict later fault and failure data, and metrics for fault prediction [6].

Our previous research works, which were based on data extracted from a large NASA mission with over two millions lines of code, were focused on characterizing and quantifying the relationships among faults, failures and fixes. The results showed that software failures were often associated with faults spread across multiple files [7]. The results further showed that a significant number of software failures required fixes in multiple software components and/or multiple software artifacts, and that the combinations of software components that were fixed together were affected by the software architecture [8]. In addition, we studied the types of faults, activities taking place when faults were detected or failures were reported, and the severity of failures [9]. The results showed that both post-release and safety-critical failures were more heavily associated with coding faults than with any other type of faults. We also analyzed the fix implementation effort and proposed a data mining approach for predicting its levels [10].

Another empirical study based on space mission data, conducted by Grottke et al., analyzed 520 anomalies from the flight software of eighteen JPL space missions and reported that 61% of bugs were Bohrbugs (i.e., bugs that are easily isolated and removed during software testing) and 37% were Mandelbugs (i.e., bugs that behave chaotically) [11]. In a follow up work, Alonso et al. analyzed the mitigation associated with the Bohrbugs and Mandelbugs [12]. Then, based on the analysis of bug reports of four open-source software systems, Cotroneo at al. [13] classified software bugs as Bohrbugs, non-aging-related Mandelbugs, and aging-related bugs.

The information extracted from bug tracking systems of open source applications was used for research with different goals. For example, Duraes and Madeira [14] explored the bug reports of 12 open source programs and classified a total of 668 faults using the Orthogonal Defect Classification (ODC) [15] with a goal to establish the fault representativeness for software fault injection experiments. Xia et al. utilized the bug tracking systems and code repositories of four open source software applications, having from 151 to 250 bug reports, which were classified into several fault categories [16].

Other papers explored software faults for different application domains. Gashi et al. studied the bug reports of four off-the-shelf SQL servers, with a focus on common faults among them, and concluded that diverse redundancy would be effective for tolerating design faults [17]. Maji et al. utilized bug reports, bug fixes, developer reports, and failure reports to explore the manifestation of failures in Android and Symbian [18]. Ocariza et al. studied the error messages printed by JavaScript as it executes in popular websites [19] and explored JavaScript faults based on 317 bug reports extracted from 12 bug repositories [20]. Frattini et al. analyzed 146 bug reports from the open source cloud platform Apache Virtual Computing Lab [21] and identified the components where bugs were likely to be found, the lifecycle phases during which such bugs may be discovered,

and the modification required to fix them. Di Martino et al. analyzed the failures of the Blue Waters, the Cray hybrid (CPU/GPU) supercomputer, and found that software was the largest contributor to the node repair hours (53%), even though it caused only 20% of the total number of failures [22].

Next, we discuss the research works that considered software vulnerabilities, either implicitly or explicitly. Several papers were focused on exploring the effectiveness of different techniques and/or tools for detection of software vulnerabilities. Specifically, Austin et al. used three electronic health record systems as case studies with a goal to compare four vulnerability discovery techniques [23]. Other research works were focused on web services domain and tested the tools' effectiveness for detection of vulnerabilities related to SQL injection, XPath injection, or XSS attacks [24], [25], [26], [27]. Another work studied the characteristics of benign, vulnerable, and malicious browser extensions and proposed an approach to detect vulnerable and malicious browser extensions [28]. Note that these works [23], [24], [25], [26], [27], [28] only implicitly addressed the types of vulnerabilities, from the perspective of tools' detection capabilities.

Two works [29], [30] explored aspects of operating systems' vulnerabilities based on the Common Vulnerabilities and Exposures (CVE) information [31]. Consequently, these works were based on vulnerabilities that were discovered post-release. In particular, Alhazmi et al. considered both commercial and open-source OSes and discovered that the vulnerability densities fell within a range of values and that the vulnerability discovery can be modeled using a logistic model [29]. Similarly, Garcia et al. utilized the CVE data to study the vulnerabilities of eleven OSes with a goal to check how many of these vulnerabilities occur in more than one OS [30].

The security vulnerabilities published in the Bugtraq database and as CERT advisories were analyzed by Chen et al. in [32] and [33], respectively. Specifically, out of the twelve classes used to classify 5,925 Bugtraq reports on software related vulnerabilities, five classes dominated: input validation errors (23%), boundary condition errors (21%), design errors (18%), failure to handle exceptional conditions (11%), and access validation errors (10%) [32]. The results of the data analysis were then combined with the source code examination to develop finite state machine (FSM) models that can be used to reason about security vulnerabilities. In a closely related work, the analysis of 107 CERT advisories showed that vulnerabilities of the following four types dominated: buffer overflow (44%), integer overflow (6%), heap corruption (8%), and format-string vulnerabilities (7%) [33]. The authors then proposed detection and/or masking techniques (i.e., memory layout randomization, control data randomization, and static analysis approach based on the notion of pointer taintedness).

Two related works that had similar main goal as ours – to study the security vulnerabilities – were focused on the web application domain [34], [35]. Fonseca and Vieira [34] explored 655 XSS and SQL injection security patches of six widely used web applications and classified the faults in each patch according to ODC [15]. The results showed that 76% of all faults found were of the Missing Function Call Extended (MFCE) type, which belongs to the ODC type 'Algorithm'. This high value was due to the massive use of specific functions to validate and clean data that come from the outside of the web applications. In a follow-up study, also focused on XSS and SQL injection, Fonseca et al. analyzed the source code of security patches of widely used web applications written in weak and strong typed languages [35]. The analysis of the weak typed language applications was based on the results presented in [34] and showed that XSS and SQL injections had similar percentages of MFCE type faults. For the strong typed language applications, the authors analyzed a sample of 60 XSS and SQL injection vulnerabilities from 11 web applications developed in Java, C#, and VB. MFCE was again the most frequent fault type, accounting for around 63% of the total number of faults. In this case, however, most of these faults (i.e., around 89%) were related to XSS vulnerabilities. Concerning SQL injection vulnerabilities, no single fault type dominated; MFCE with 21% and MIEB (i.e., Missing if construct plus statements plus else before statements) with 26% were the two most frequent fault types, followed by several other fault types.

III. Classification approach

In order to classify the security related bugs (i.e., vulnerabilities), a classification schema is needed. An obvious candidate for a classification schema is the Common Weakness and Enumeration (CWE) taxonomy of software weakness types, which serves as a common language for describing software security weaknesses in architecture, design, or code [36]. Each individual CWE represents a single vulnerability category. CWEs are organized in a hierarchical structure with broad category CWEs at the top level. The further down this hierarchy, the more specific the vulnerabilities become. The CWE taxonomy has 1006 CWEs and rather complex structure; each CWE may have one or more parents (expect the top level CWEs) and zero or more children. Therefore, using the complete CWE taxonomy for classification of software vulnerabilities is not very practical, which is the reason why a number of views have been developed to ease the grouping of similar CWEs and provide simpler structures. These include: CWE-1000 [37], CWE-888 [4], [5], CWE-700 [38], [39], and CWE-699 [40].

Upon close review, we selected CWE-888 Software Fault Pattern (SFP) View as classification schema because it provides intuitive hierarchical structure, with a good trade-off between the level of details and generality. CWE 888

Table I. PRIMARY CLASSES, THEIR DEFINITIONS AND THE CORRESPONDING SECONDARY CLASSES

Primary class	**Definition and corresponding secondary classes**
Risky Values	Relates to the basic uses of numerical values in software systems. Secondary class: Glitch in Computation.
Unused Entities	Covers unused entities in code, including unused procedures or variables. Secondary class: Unused Entities.
API	Relates to the use of Application Programming Interfaces (API). Secondary class: Use of an Improper API
Exception Management	Relates to management of exceptions and other status conditions. Secondary classes: Unchecked Status Condition, Ambiguous Exception Type, and Incorrect Exception Behavior.
Memory Access	Relates to access to memory buffers. Secondary classes: Faulty Pointer Use, Faulty Buffer Access, Faulty String Expansion, Incorrect Buffer Length Computation, Improper NULL termination.
Memory Management	Relates to the management of memory buffers. Secondary classes: Faulty Memory Release.
Resource Management	Relates to management of resources (i.e., dynamic entities). Secondary classes: Unrestricted Consumption, Failure to Release Resource, Faulty Resource Use, Life Cycle.
Path Resolution	Relates to access to file resources using complex file names. Secondary classes: Path Traversal, Failed Chroot Jail, Link in Resource Name Resolution
Synchronization	Relates to the use of shared resources. Secondary classes: Missing Lock, Race Condition Window, Multiple Locks/Unlocks, Unrestricted Lock.
Information Leak	Relates to the export of sensitive information from an application. Secondary classes: Exposed Data, State Disclosure, Exposure Through Temporary File, Other Exposures, Insecure Session Management.
Tainted Input	Relates to injection of user controlled data into various destination commands. Secondary classes: Tainted Input to Command, Tainted Input to Variable, Composite Tainted Input, Faulty Input Transformation, Incorrect Input Handling, Tainted Input to Environment.
Entry Points	Relates to unexpected entry points into the application. Secondary class: Unexpected Access Points.
Authentication	Relates to establishing the identity of an actor associated with the computation, or the identity of the endpoint involved in the computation through a certain channel. Secondary classes: Authentication Bypass, Faulty Endpoint Authentication, Missing Endpoint Authentication, Digital Certificate, Missing Authentication, Insecure Authentication Policy, Multiple Binds to the Same Port, Hardcoded Sensitive Data, Unrestricted Authentication.
Access Control	Relates to validating resource owners and their permissions. Secondary classes: Insecure Resource Access, Insecure Resource Permissions, Access Management.
Privilege	Relates to code regions with inappropriate privilege level. Secondary class: Privilege.
Channel	Relates to various protocol issues. Secondary classes: Channel Attack, Protocol Error.
Cryptography	Relates to cryptography issues. Secondary classes: Broken Cryptography, Weak Cryptography.
Malware	Relates to any malicious code present in the software system. Secondary classes: Malicious Code, Covert Channel.
Predictability	Relates to random number generators and their properties. Secondary class: Predictability.
UI	Relates to security issues of User Interfaces (UI). Secondary classes: Feature, Information Loss, Security.
Other	Relates to miscellaneous architecture, design, and implementation issues. Secondary classes: Architecture, Design, Implementation, Compiler.

contains 705 CWEs organized in a three level hierarchical structure, of which the first two levels (primary and secondary classes) were used for our classification. Namely, we assigned each security related software bug to a primary (more general) class and the corresponding secondary (more specific) class. Overall, there are 21 primary and 62 secondary classes (see Table I). More detailed descriptions and the specific CWE numbers can be found in [4].

We conducted manual classification of software bug reports in all three datasets using the information provided in the 'Title', 'Subject', 'Description', 'Recommended Actions', and 'Solution' fields from the issue tracking systems. Similarly to the classification done by static code analysis tools, we adopted a conservative classification approach that treats as security related every bug report that can be assigned a CWE-888 class. For example, the bug report with description "...The stream is opened on line 603 of file1. If an exception were to occur at any point before line 613 where it is closed, then the 'try' would exit and the stream would not be closed," was classified as the primary class 'Resource Management' and the secondary class 'Failure to Release Resource.' On the other side, the bug report with description "...Table 1-11 lists XYZ as a unidirectional interfaces, but Figure 1-4 shows this connection as bidirectional," was classified as non-security related.

Note that we did not have access to the code and other information needed to determine if the security issues (i.e., vulnerabilities) could be easily exploited or what the overall impact on the system would be if a vulnerability was successfully exploited. Therefore, these aspects are out of the scope of our work.

IV. DESCRIPTION OF THE DATASETS

The three datasets used for this work were created by extracting relevant information from the issue tracking systems of two NASA missions. For all three datasets, only the issues that were marked as bug reports and were closed were included in the analysis.

The first dataset was extracted from the IV&V issue tracking system of a NASA ground mission and is referred to as *Ground mission IV&V issues*. The ground mission software consists of approximately 1.36 million source lines of code and the issue tracking system contained 1,779 issues created over four years. Since this is a recent mission, the IV&V analysts specifically considered the impact of each issue on security, and as a result 350 of the issues were marked as security related and their descriptions contained security

related information. Based on the manual classification, we assigned specific CWEs to 133 of the 350 security issues. The remaining security issues were tagged by the IV&V analysts as testing issues. Since testing issues do not deal with the actual software under investigation and no CWEs exist that cover such issues, testing issues were excluded from the further analysis. (Note that the ground mission developers' issues were not available to the research team.)

The second dataset consists of the IV&V issues extracted from the issue tracking system of a NASA flight mission and is referred to as *Flight mission IV&V issues*. The flight mission software had approximately 924 thousand source lines of code, and the issue tracking system contained 383 issues. It should be noted that the IV&V issues of the Flight Mission neither were tagged as security related by the IV&V analysts nor the security aspects of the issues were specifically and consistently addressed in the issues' descriptions. We manually classified these 383 issues, out of which 157 issues appeared to be security related.

The third dataset consists of issues extracted from the developers' issue tracking system of the same NASA flight mission and is referred to as *Flight mission Developers issues*. This issue tracking system contained 573 closed bug reports. As in case of the Flight mission IV&V issues dataset, security aspects of developers' issues were not specifically addressed in the descriptions and issues were not tagged as security / not security related. Based on manual classification, we marked 374 bug reports (out of 573) as security related and assigned them specific CWE classes.

The basic facts of the two missions and the three datasets are summarized in Table II.

Table II. BASIC FACTS ABOUT THREE DATASETS

Mission	Size	Total # closed bug reports	Security related bug reports	Dataset
Ground	1.36 MLOC	1,779	133	Ground mission IV&V
Flight	924 KLOC	383	157	Flight mission IV&V
		573	374	Flight mission Developers

V. VULNERABILITY PROFILES

In this section we present the results for each of the three datasets. (As described in section IV, the term 'issue' refers to 'closed bug reports'.)

A. Ground Mission IV&V Issues

Figure 1 shows the distribution of issues (i.e., bug reports) across different Issue Categories. As shown, the Code category contained 95% of all security issues. Even though the Design category had the highest number of issues, only around 2% of security issues belonged to this category. Figure 2 shows the distribution of issues across different Issues Types, which provide more detailed categorization than the Issue Category. Two most dominant security issue types were Incomplete Code and Incorrect Code, which together contained 84% of all security related issues.

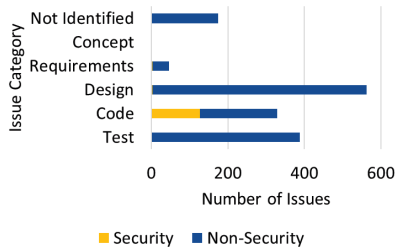

Figure 1. Issue Categories of the Ground mission IV&V issues

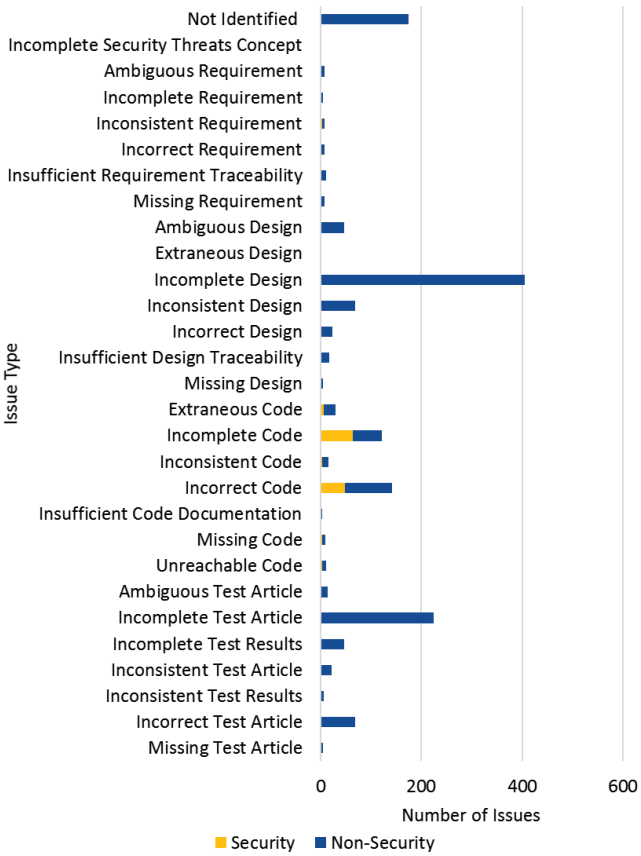

Figure 2. Issue Types of the Ground mission IV&V issues

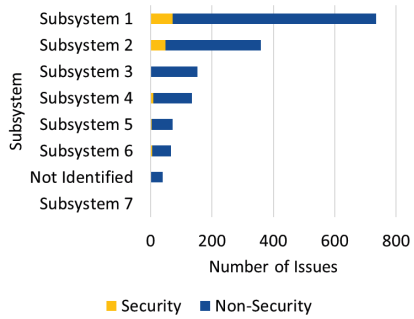

Figure 3. Distribution of Ground mission IV&V issues across subsystems

Figure 3 shows the breakdown of issues across subsystems, ordered from the subsystem with the highest total

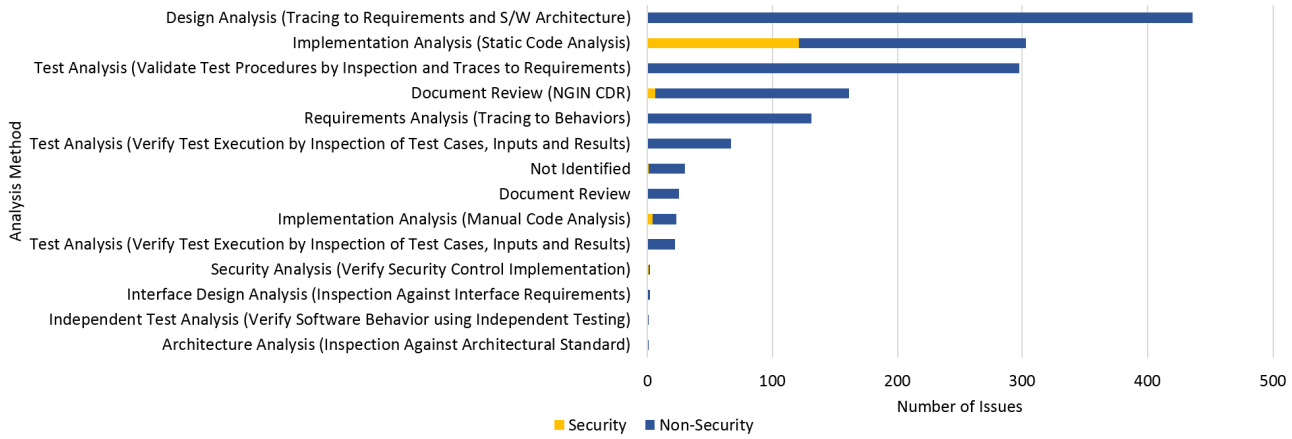

Figure 4. Distribution of Ground mission IV&V issues across Analysis Methods

number of issues to the subsystem with the least issues. Subsystems 1 and 2 contributed 86% of all security issues and 70% of all issues, which shows that Pareto principle[1] applies to security related issues, as well as to the total number of issues.

Figure 4 shows the distribution of issues with respect to the analysis method used to detect the issues. The largest proportion of total issues (30%) was found using Design Analysis; however this method did not uncover any security issues. The vast majority of security issues were discovered using Implementation Analysis (Static Code Analysis). Specifically, this method led to finding 91% of all security related issues[2]. It should be noted that the amount of time and effort invested in using each Analysis Method affect the number of issues (including security issues) detected by that method. Unfortunately, the time and effort used for each Analysis Method were not tracked, and therefore we cannot draw conclusions about the effectiveness of the Analysis Methods based on the results presented in Figure 4.

The distribution of issues across different severity levels is shown in Figure 5. NASA's Severity levels range from 1 to 5, with 1 being the most severe. As shown in Figure 5, 86% of all security related issues had severity level 3, as well as the majority of all issues (72%).

Figures 6 and 7 detail the phase in which issues were introduced and found, respectively. (Note that this mission is under development and has not entered the Test phase yet.) The majority of security issues (91%) were introduced in the Implementation Phase, which indicates how hard it is to implement secure code. This result also shows that efforts to

[1]Pareto principle indicates skewed distribution of software faults, that is, that majority of faults (e.g., roughly 80%) are located in small percent (e.g., 20%) of software units (e.g., subsystems or files.)

[2]Static code analysis tools are known to produce high number of false positives [41]. In this case the output produced by the static code analysis tool was manually inspected by the IV&V analysts and only true positive warnings were entered as bug reports in the issue tracking system.

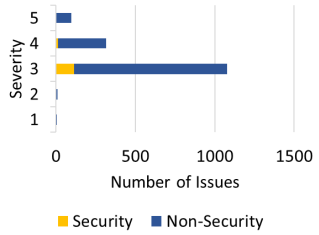

Figure 5. Severity levels of Ground mission IV&V issues

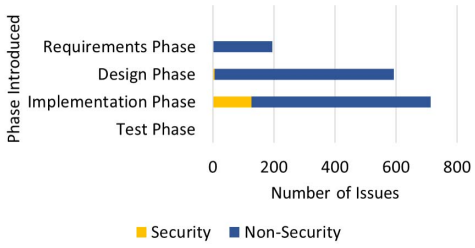

Figure 6. Ground mission IV&V issues: Phase Introduced

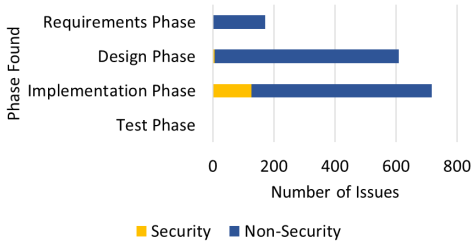

Figure 7. Ground Mission IV&V issues: Phase Found

enforce secure coding standards would lead to cost effective improvement of mission's security. Comparing Figures 6 and 7 can be observed that the phase in which issues were found closely followed the phase in which they were introduced, which illustrates the effectiveness of the IV&V activities.

Next, we focus on the distribution of the security related issues across the primary classes shown in Table I. As

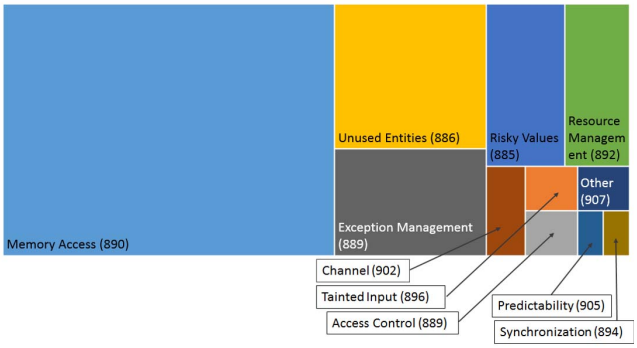

Figure 8. Ground mission IV&V issues: Distribution across primary classes. The numbers in brackets represent the specific CWE numbers of the primary classes.

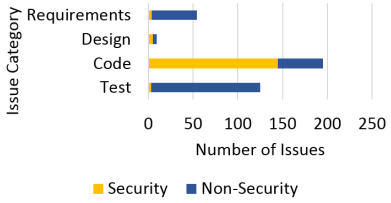

Figure 9. Issue Categories of Flight mission IV&V issues

can be seen in Figure 8, the IV&V security issues of the ground mission belonged to only 11 out of the total 21 primary classes. The Memory Access dominated, containing 54.6% of all security issues. Furthermore, only five primary classes (i.e., Memory Access, Unused Entities, Exception Management, Risky Values, and Resource Management) contained around 95% of all security issues. This result shows that the Parato principle applies to the distribution of the security issues across primary classes as well.

B. Flight Mission IV&V Issues

As shown in Figure 9, 92% of all security related issues were associated with the Code Issue Category. This distribution of security related issues is consistent with the results for the Ground mission IV&V issues. Figure 10 shows the distribution of Flight mission IV&V issues across Issue Types, which provide more detailed information than the Issue Category. The results show that security issues were predominately associated with Incorrect Code, Incomplete Code, Missing Code, Extraneous Code, and Inconsistent Code.

The distribution of issues across Flight mission subsystems presented in Figure 11 shows that 88% of all security issues (and 88% of all issues) fell into three out of five subsystems. (Subsystems in Figure 11 are ordered by the total number of issues.)

As shown in Figure 12, severity levels 3 and 4 together contained 79% of all security issues and 86% of all issues. The fact that not many security issues had high Severity levels (i.e., 1 and 2) is consistent with the Ground mission IV&V security issues.

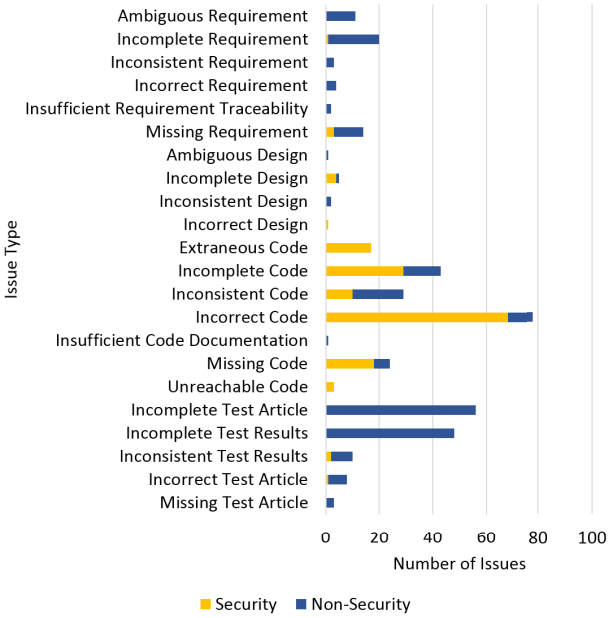

Figure 10. Issue Types of Flight mission IV&V issues

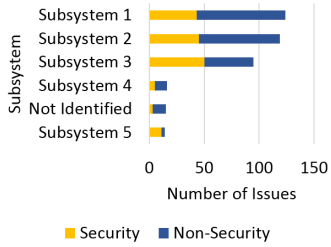

Figure 11. Distribution of Flight Mission IV&V issues across Subsystems

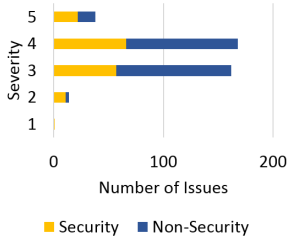

Figure 12. Severity levels of Flight Mission IV&V issues

Figures 13 and 14 also show results consistent with the Ground Mission IV&V issues, with the majority of security issues introduced (85%) and found (85%) in the Implementation phase. The Flight Mission IV&V Issues dataset, in addition, included information on the phase in which the issues were resolved. As can be seem in Figure 15, 75% of security related issues were resolved in the Implementation phase, and the remaining 25% were resolved in the Testing phase.

Next, we focus on the distribution of security issues across the primary classes, which is presented in Figure 16. Similarly as in the case of the Ground Mission IV&V Issues,

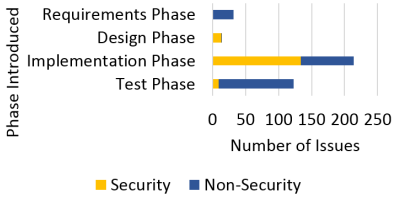

Figure 13. Flight Mission IV&V issues: Phase Introduced

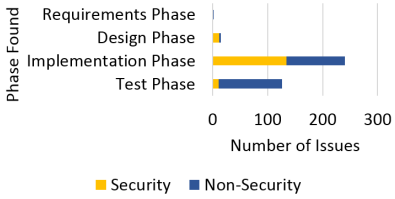

Figure 14. Flight Mission IV&V issues: Phase Found

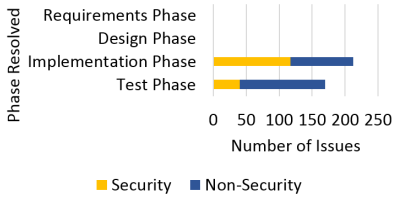

Figure 15. Flight Mission IV&V issues: Phase Resolved

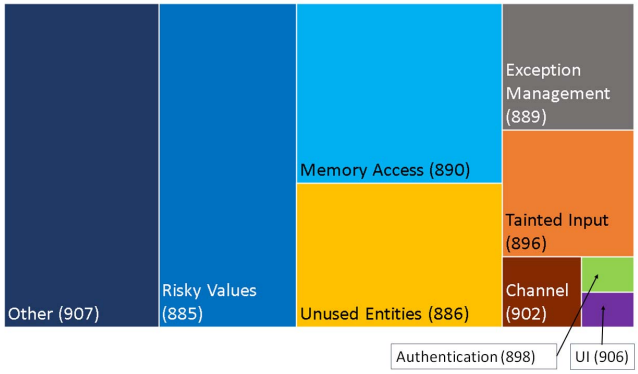

Figure 16. Flight Mission IV&V issues - Distribution of issues across primary classes. The numbers in brackets represent the specific CWE numbers of the primary classes.

IV&V issues of the Flight mission belonged to only 9 out of the 21 primary classes, with four dominant classes: Other, Risky Values, Memory Access, and Unused Entities.

C. Flight Mission Developers Issues

The developers' issue tracking system of the flight mission did not contain the Issue Category field. Figure 17 shows the distribution of issues across Issue Types. The two Issues Types – 'Incorrect Operation or Unexpected Behavior' and 'Incorrect Implementation' – significantly outnumbered the other Issue Types.

Figure 18 presents the distribution of issues across the Flight mission subsystems, ordered by the total number of issues. The finding is similar to the previous datasets,

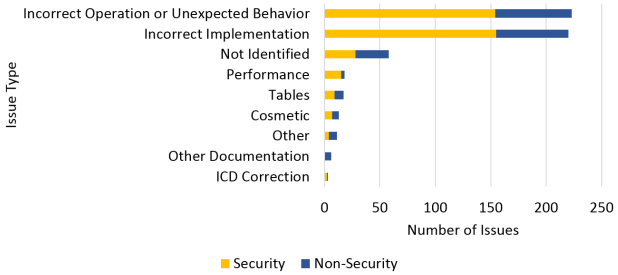

Figure 17. Issue Types of Flight mission Developers issues

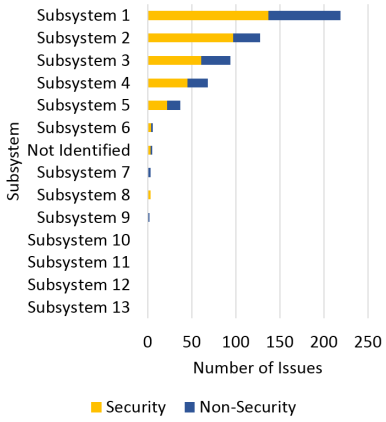

Figure 18. Distribution of Flight mission Developers issues across Subsystems

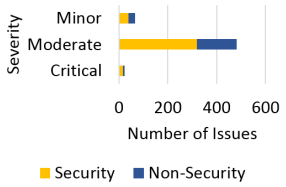

Figure 19. Severity levels of Flight mission Developers issues

again proving the Pareto principle, with 88% of all security issues found in only four subsystems (out of thirteen), which together accounted for 89% of all issues. (While more subsystems appeared in the Developers issue tracking system than in the IV&V issue tracking system of the flight mission, the three most fault prone subsystems in both Figures 11 and 18 are the same.)

The severity levels used in this dataset were: Minor, Moderate, and Critical. As shown in Figure 19, the results related to the severity of the Flight mission Developers issues were consistent to the previously analyzed datasets; the moderate severity level dominated, containing 86% of the security issues, and 85% of the total number of issues. Only 4% of all issues, and 4% of security issues were determined to be critical.

This dataset contained information about the phase in which the issues were found, but no information on when they were introduced or resolved. As shown in Figure 20, most issues were found during the following there phases:

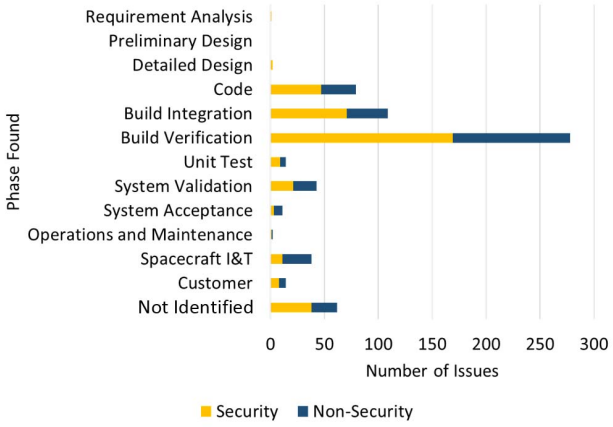

Figure 20. Flight mission Developers issues: Phase Found

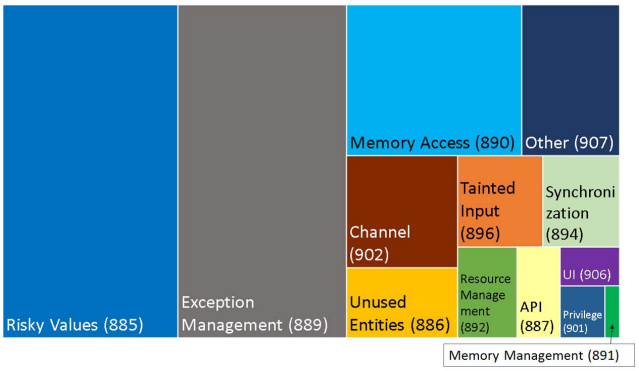

Figure 21. Flight mission Developers issues - Distribution across primary classes. The numbers in brackets represent the specific CWE numbers of the primary classes.

Code Implementation, Build Integration, and Build Verification. These more fine grained phases are consistent with the Phase Found categories that dominated the IV&V issues.

Next, we focus on the classification of the security issues. Similarly as for the other two datasets, as shown in Figure 21, only 13 out of 21 primary class were observed, with three dominant classes: Risky Values, Exception Management, and Memory Access.

VI. Comparison of the Results Across Datastes

In this section we compare the results across all three datasets. We start with comparing the distribution of security issues across the primary classes, extracted from the results presented in subsections V-A, V-B, and V-C. As can be seen in Table III even though fifteen (out of 21) primary classes had nonzero security issues for at least one dataset, the vast majority of security issues were distributed among five dominant primary classes: Exception Management, Memory Access, Other, Risky Values, and Unused Entities. Specifically, these five primary classes together contained 90%, 87%, and 79% of the security issues in the Ground mission IV&V, Flight mission IV&V and Flight mission Developers issues, respectively.

Table III. Comparison of primary classes (with nonzero security issues) across the three datasets. The five dominant classes are shared gray.

Primary class	Ground Mission	Flight Mission	
	IV&V Issues	**IV&V Issues**	**Developer Issues**
API (887)			1.9%
Authentication (898)		0.9%	
Channel (902)		2.7%	6.0%
Exception Management (889)	10.8%	8.2%	27.2%
Memory Access (890)	54.6%	18.3%	12.8%
Memory Management (891)			0.4%
Other (907)	1.5%	24.5%	7.1%
Predictability (905)	0.8%		
Privilege (901)			1.2%
Resource Management (892)	6.9%		3.0%
Risky Values (885)	8.5%	21.8%	28.3%
Synchronization (894)	0.8%		3.4%
Tainted Input (896)	1.5%	8.2%	3.8%
UI (906)		0.9%	1.1%
Unused Entities (886)	14.6%	14.5%	3.8%

Table IV. Secondary classes, for the five dominant primary classes

Secondary class	Ground Mission	Flight Mission	
	IV&V Issues	IV&V Issues	Developer Issues
Exception Management (889)			
Ambiguous Exception Type (960)	7.7%		
Incorrect Exception Behavior (961)		4.6%	14.0%
Unchecked Status Condition (962)	3.1%	3.6%	13.2%
Memory Access (890)			
Faulty Buffer Access (970)	4.6%	12.7%	8.3%
Faulty Pointer Use (971)	50.0%	5.6%	4.5%
Other (907)			
Architecture (975)		0.9%	
Design (977)			2.6%
Implementation (978)	1.5%	23.6%	4.5%
Risky Values (885)			
Glitch in Computation (998)	8.5%	21.8%	28.3%
Unused Entities (886)			
Dead Code (561)	14.6%	10.0%	3.4%
Unused Variable (563)		4.5%	0.4%

Table IV shows the five dominant primary classes along with their corresponding secondary classes, which provide more detailed information on the nature of security issues. The secondary classes under the Exception Management primary class included: Ambiguous Exception Type, Incorrect Exception Behavior, and Unchecked Status Condition. Note that 'Failure to handle exceptional conditions' was among dominant classes of Bugtraq vulnerabilities analyzed in [32].

Security issues assigned to the primary class Memory Access were distributed across the secondary classes Faulty Buffer Access and Faulty Pointer Use. These categories include common programming errors such as null pointer dereferences and buffer overflows. Not surprisingly, faulty memory access (including buffer overflows) were among prevalent vulnerability classes in both [32] (belonging to the boundary condition errors) and [33] (belonging to the buffer overflows and heap corruption classes).

The primary class Other had security issues distributed predominately in the secondary class Implementation, which is based around weaknesses such as coding standards violation or containment errors. The secondary classes Architecture and Design had significantly less issues. (Note that issues were assigned to the primary class Other and the

Table V. MAIN FINDINGS ABOUT SECURITY ISSUES ACROSS ALL DATASETS

	Ground mission IV&V issues	Flight mission IV&V issues	Flight mission Developers issues	RQ
Issue category	95% Code related	92% Code related	Data not available	RQ1
Subsystem	86% found in two subsystems (70% of all issues)	88% in three subsystems (88% of all issues)	88% in four subsystems (89% of all issues)	RQ2
Phase Introduced	91% in the Implementation Phase	85% in the Implementation Phase	Data not available	RQ3
Phase Found	Followed closely the phase introduced distribution	Followed closely the phase introduced distribution	Most found during Code Implementation, Build Integration, and Build Verification	
Severity	Level 3 dominated (86%)	Levels 3 and 4 dominated (79%)	Moderate dominated (86%)	RQ4
Five (out of 21) most frequent primary Classes	Exception Management 10.8% Memory Access 54.6% Other 1.5% Risky Values 8.5% Unused Entities 14.6% Total 90%	Exception Management 8.2% Memory Access 18.3% Other 24.5% Risky Values 21.8% Unused Entities 14.5% Total 87%	Exception Management 27.2% Memory Access 12.8% Other 7.1% Risky Values 28.3% Unused Entities 3.8% Total 79%	RQ5

corresponding secondary classes when they were related to these classes and could not be placed into any other class.)

The primary class Risky Values consisted of the secondary class Glitch in Computation, which deals with calculation errors such as divide by zero error or a function call with an incorrect order of arguments.

The primary class Unused Entities consisted predominately of the secondary class Dead Code and much less of the secondary class Unused Variable.

Table V summarizes the main findings as they pertain to RQ1 - RQ5 and helps identifying trends that are consistent across datasets (i.e., RQ6).

Interestingly, the Code related security issues dominated both the Ground mission IV&V security issues and the Flight mission IV&V security issues, with 95% and 92%, respectively. We believe that this is an important finding, especially having in mind that software bugs (including the security related bugs) were discovered using different methods, throughout the life cycle (e.g., see Figure 4). This finding implies that enforcing secure coding practices and verification and validation focused on coding errors (for example by using a check list for inspection) would be cost effective ways to improve missions' security.

The location of security issues, as the location of total issues, followed the Pareto principle. Specifically, from 86% to 88% of security issues were located in two to four subsystems, for all three datasets. This result is consistent with the related works focused on characterization of software bugs in general (e.g., [6], [7], [9]).

In both the Ground mission and Flight mission IV&V issues datasets majority of issues (i.e., 91% and 85%, respectively) were introduced in the Implementation phase. This is consistent with the fact that majority of security issues were code related. It is important that software vulnerabilities (i.e., security issues) are fixed in a timely manner. The good news is that in most cases the phase in which the issues were found was the same as the phase in which they were introduced. The most security related issues of the Flight mission Developers issues dataset were found during Code Implementation, Build Integration, and Build Verification, which is consistent with the other datasets.

With respect to severity, the results showed that the security issues, as the majority of all issues, were with moderate severity, for all three datasets.

The final row in Table V lists the five dominant primary classes (out of 21 classes), which together contributed from around 80% to 90% of all security issues in each dataset. This again proves the Pareto principle of uneven distribution of security issues across CWE classes and supports the fact that addressing these dominant security classes provides the most cost efficient way to improve missions' security. Note that faulty Memory Access was among prevalent types of vulnerabilities in [32], [33], and a significant percentage of Exception Management vulnerabilities were reported in [32]. Furthermore, the related works which explored software vulnerabilities in web applications [34], [35] also found skewed distributions across vulnerability classes. However, since these two works were focused on a different domain (i.e., web applications) and analyzed only XSS and SQL injections, detailed comparison of vulnerability classes with our work is not feasible. In general, comparing dominant vulnerability classes across related works is a challenging task because different works used different classification schemas.

VII. THREATS TO VALIDITY

In this section we discuss the threats to validity to our study. **Construct validity** is concerned with whether we are measuring what we intend to measure. The number and classes of identified security issues depend on the quality of software artifacts, as well as the level of provided details related to security. In the case of the Ground mission IV&V dataset, significant number of security issues were tagged as testing related. Since no CWE exists that covers such cases and testing issues are not related to software itself, these testing related security issues were not included in the further analysis. Another threat to construct validity is related to the fact that some security issues could be correctly classified into multiple CWE classes. However,

the number of issues fitting into multiple CWE classes was small, and for these cases the most relevant of the possible classes was selected.

Internal validity threats are concerned with unknown influences that may affect the independent variables. Data quality is one of the major concerns to the internal validity. It should be noted that NASA issue tracking systems follow high record keeping standards, which provides some guarantee for the quality and consistency of the data. As mentioned in section IV, in the case of the Flight mission datasets (both the IV&V issue and Developers issues) software issues were not tagged as security related and security aspects of software bugs were not explicitly addressed in the descriptions. Therefore, it is possible that some potential security aspects were not reflected in the available information and, therefore, could not be accounted for in our analysis.

Conclusion validity threats impact the ability to draw correct conclusions. One threat to conclusion validity is related to data sample sizes. The work presented in this paper is based on three dataset, with a total of 2,735 issues, out of which 664 were classified as security related. The numbers of security issues per dataset were also large, ranging from 133 to 374. The results and conclusions may be affected by the fact that the types of security issues (and consequently the identified primary and secondary classes) may depend on the validation and verification (V&V) methods used, as well as the amount of time and effort expended on using these methods. The analysis methods in our case were explicitly available for only one of the datasets. However, we confirmed with the NASA personnel that the V&V and IV&V activities for the three datasets spanned the whole software lifecycle and were focused on different software artifacts, including requirements, design, and code.

External validity is concerned with the ability to generalize results. The breadth of this study, including the facts that (1) it is based on two large NASA missions containing around one million lines of code each and (2) the missions were developed by different teams over multiple years, allow for some degree of external validation. Nevertheless, we cannot claim that the results would be valid for other mission critical software systems. Furthermore, one can expect different application domains to have different sets of dominant vulnerability classes. Therefore, the external validity remains to be established by similar future studies that will use other software products as case studies.

VIII. Conclusion

While prior empirical work on characteristics of software faults (i.e., bugs) and failures exists, much less research works were focused on analyses of software application vulnerabilities. This paper contributes towards building an evidence-based knowledge about different aspects of software application vulnerabilities. The empirical findings presented in the paper are based on the data extracted from the issue tracking systems of two NASA missions. It should be noted that this study accounts for vulnerabilities that may have been introduced, found, and fixed throughout the software lifecycle. Using the extracted data, organized in three datasets, we built so called security vulnerability profiles that address several aspects of software application vulnerabilities, such as where and when the security vulnerabilities were introduced and what were the dominant vulnerabilities classes. An important aspect of this paper is the identification of trends that are consistent across the three datasets.

The main findings of this work indicate that the majority of vulnerabilities were code related and were introduced in the Implementation phase, and that the Pareto principle (i.e., uneven distributions) applied both to the location of vulnerabilities across subsystems and to the distribution across different vulnerability classes. It appears that development and testing efforts focused on these vulnerability prone subsystems and dominant vulnerability classes provide the most cost efficient ways to improve missions' security.

We believe that mining the information related to software vulnerabilities supports building a knowledge base that could be useful for other similar systems. Exploring the same research questions on other case studies (from the space and other domains) would help establishing the characteristics of software application vulnerabilities that are invariant across different software systems and/or application domains.

Acknowledgments

This work was funded in part by the NASA Software Assurance Research Program (SARP) in 2016 fiscal year. Any opinions, findings, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agency. The authors thank the following NASA personnel for their support: Brandon Bailey, Craig Burget, and Dan Painter.

References

[1] "NASA cybersecuirty presentation," NASA Office of the Chief Information Officer, Nov 2014.

[2] K. Osborn, "Air force faces increasing space threats: Shelton," in *DefenseTech*, Sep 2013.

[3] "Inadequate security practices expose key NASA network to cyber attack," Office of Inspector General, Audit report, May 2011.

[4] N. Mansourov, "Software fault patterns (SFP)," 2011, [online] https://samate.nist.gov/BF/Enlightenment/SFP.html.

[5] "CWE-888: Software fault pattern (SFP) clusters, MITRE Corporation," https://cwe.mitre.org/data/graphs/888.html.

[6] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797–814, Aug 2000.

[7] M. Hamill and K. Goseva-Popstojanova, "Common trends in software fault and failure data," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 484–496, July 2009.

[8] ——, "Exploring the missing link: An empirical study of software fixes," *Software Testing, Verification and Reliability*, vol. 24, no. 8, pp. 684–705, 2014.

[9] ——, "Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system," *Software Quality Journal*, vol. 23, no. 2, pp. 229–265, 2015.

[10] ——, "Analyzing and predicting effort associated with finding and fixing software faults," *Information and Software Technology*, vol. 87, pp. 1–18, 2017.

[11] M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," in *40th IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, June 2010, pp. 447–456.

[12] J. Alonso, M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault repairs and mitigations in space mission system software," in *43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2013, pp. 1–8.

[13] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, "Fault triggers in open-source software: An experience report," in *24th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2013, pp. 178–187.

[14] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Transactions of Software Engineering*, vol. 32, no. 11, pp. 849–867, Nov. 2006.

[15] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal Defect Classification – A concept for in-process measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943–956, Nov. 1992.

[16] X. Xia, X. Zhou, D. Lo, and X. Zhao, "An empirical study of bugs in software build systems," in *13th International Conference on Quality Software*, July 2013, pp. 200–203.

[17] I. Gashi, P. Popov, and L. Strigini, "Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 4, pp. 280–294, 2007.

[18] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi, "Characterizing failures in mobile OSes: A case study with Android and Symbian," in *21st IEEE International Symposium on Software Reliability Engineering*, Nov 2010, pp. 249–258.

[19] F. S. Ocariza, K. Pattabiraman, and B. Zorn, "JavaScript errors in the wild: An empirical study," in *22nd IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2011, pp. 100–109.

[20] F. S. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side JavaScript bugs," in *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 55–64.

[21] F. Frattini, R. Ghosh, M. Cinque, A. Rindos, and K. S. Trivedi, "Analysis of bugs in Apache Virtual Computing Lab," in *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2013, pp. 1–6.

[22] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and J. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of Blue Waters," in *44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014, pp. 610–621.

[23] A. Austin, C. Holmgreen, and L. Williams, "A comparison of the efficiency and effectiveness of vulnerability discovery techniques," *Information and Software Technology*, vol. 55, no. 7, pp. 1279–1288, 2013.

[24] J. Fonseca, M. Vieira, and H. Madeira, "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.

[25] N. Antunes and M. Vieira, "Comparing the effectiveness of penetration testing and static code analysis on the detection of SQL injection vulnerabilities in web services," in *15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2009, pp. 301–306.

[26] N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira, "Effective detection of SQL/XPath injection vulnerabilities in web services," in *IEEE International Conference on Services Computing*, 2009, pp. 260–267.

[27] M. Vieira, N. Antunes, and H. Madeira, "Using web security scanners to detect vulnerabilities in web services," in *39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2009, pp. 566–571.

[28] H. Shahriar, K. Weldemariam, M. Zulkernine, and T. Lutellier, "Effective detection of vulnerable and malicious browser extensions," *Computer & Security*, vol. 47, pp. 66–84, 2014.

[29] O. Alhazmi, Y. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *Computers & Security*, vol. 26, no. 3, pp. 219 – 228, 2007.

[30] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro, "Analysis of operating system diversity for intrusion tolerance," *Software – Practice and Experience*, vol. 44, no. 6, pp. 735–770, 2014.

[31] "Common Vulnerabilities and Exposures (CVE)," January 2017, https://cwe.mitre.org/.

[32] S. Chen, Z. Kalbarczyk, J. Xu, and R. Iyer, "A data-driven finite state machine model for analyzing security vulnerabilities," in *IEEE International Conference on Dependable Systems and Networks*, 2003, pp. 605–614.

[33] S. Chen, J. Xu, Z. Kalbarczyk, and R. Iyer, "Security vulnerabilities: From analysis to detection and masking techniques," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 407–418, 2006.

[34] J. Fonseca and M. Vieira, "Mapping software faults with web security vulnerabilities," in *38th IEEE International Conference on Dependable Systems and Networks (DSN)*, 2008, pp. 257–266.

[35] J. Fonseca, N. Seixas, M. Vieira, and H. Madeira, "Analysis of field data on web security vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 2, pp. 89–100, 2014.

[36] "Common Weakness Enumeration (CWE)," January 2017, https://cwe.mitre.org/.

[37] "CWE-1000: Research concepts, MITRE Corporation," https://cwe.mitre.org/data/graphs/1000.html.

[38] K. Tsipenyuk, B. Chess, and G. McGraw, "Seven pernicious kingdoms: A taxonomy of software security errors," *IEEE Security Privacy*, vol. 3, no. 6, pp. 81–84, Nov 2005.

[39] "CWE-700: Seven pernicious kingdoms, MITRE Corporation," https://cwe.mitre.org/data/definitions/700.html.

[40] "CWE-699: Development concepts, MITRE Corporation," https://cwe.mitre.org/data/graphs/699.html.

[41] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Information and Software Technology*, vol. 68, no. C, pp. 18–33, Dec. 2015.