# PREVENTING AND DIAGNOSING SOFTWARE UPGRADE FAILURES

**BY REKHA BACHWANI**

**A dissertation submitted to the**

**Graduate School—New Brunswick**

**Rutgers, The State University of New Jersey**

**in partial fulfillment of the requirements**

**for the degree of**

**Doctor of Philosophy**

**Graduate Program in Computer Science**

**Written under the direction of**

**Ricardo Bianchini**

**and approved by**

**New Brunswick, New Jersey**

**January, 2012**

## ABSTRACT OF THE DISSERTATION

# Preventing and Diagnosing Software Upgrade Failures

**by Rekha Bachwani**

**Dissertation Director: Ricardo Bianchini**

Modern software systems are complex and comprise many interacting and dependent components. Frequent upgrades are required to fix bugs, patch security vulnerabilities, and add or remove features. Unfortunately, many upgrades either fail or produce undesired behavior resulting in service disruption, user dissatisfaction, and/or monetary loss. To make matters worse, when upgrades fail or misbehave, developers are given limited (and often unstructured) information to pinpoint and correct the problems.

In this dissertation, we propose two systems to improve the management of software upgrades. Both systems rely on environment information and dynamic execution data collected from users who have previously upgraded the software. The first (called Mojave) is an upgrade recommendation system that informs a user who intends to upgrade the software about whether the upgrade is likely to succeed. Regardless of Mojave's recommendation, if the user decides to upgrade and it fails, our second system (called Sahara) comes into play. Sahara is a failed upgrade debugging system that identifies a small subset of routines that are likely to contain the root cause of the failure. We evaluate both systems using several real upgrade failures with widely used software. Our results demonstrate that our systems are very accurate in predicting upgrade failures and identifying the likely culprits for upgrade failures.

# Acknowledgements

This dissertation is a culmination of an exciting and consuming journey of discovery, learning and hard work. I would like to take this opportunity to express my gratitude to all the people who have helped me in consummating this dissertation.

First and foremost, I would like to thank my advisor, Ricardo Bianchini. His guidance, insatiable quest for perfection, perseverance, and support throughout my Ph.D have helped me complete this thesis, and become a better researcher. In addition to Ricardo, I was very fortunate to work with Willy Zwaenepoel and Dejan Kostic. I could always count on their feedback and advice for my work, despite their busy schedule and the thousands of miles separating us. Thank you Willy and Dejan! Furthermore, I am grateful to Thu Nguyen and Rich Martin for guiding me early on in my Ph.D, and providing invaluable feedback to improve my thesis towards the end as committee members.

I had the pleasure of working with many able minds during my Ph.D. I would like to thank my collaborators and co-authors Olivier Crameri, Leszek Gryz, Cezary Dubnicki, and Fabio Oliviera for working with me. In addition, I would like to extend my gratitude to my colleagues Mangesh Gupte, Inigo Goiri, Luiz Ramos, Kien Le, Wei Zheng, Ana Paula Centeno, and Qingyuan Deng for the myriad brainstorming sessions and invigorating conversations over caffeine and meal breaks. My life as a graduate student would not have been as much fun without your company and support.

Last but not the least, I would like thank my family for being there for me whenever I needed them. I am grateful to you all for your unwavering belief in me, and for being my continual source of encouragement and strength. Finally, I would like to thank my close friends, Prashant and Preethi, for letting me lean on their shoulders throughout my Ph.D.

# Table of Contents

**Abstract** . . . ii

**Acknowledgements** . . . iii

**List of Tables** . . . vi

**List of Figures** . . . vii

**List of Code Listings** . . . viii

**1. Introduction** . . . 1

**2. Bug Characterization** . . . 5

2.1. Introduction . . . 5

2.2. Methodology . . . 6

2.3. Survey Results . . . 6

2.3.1. Frequency of environment-related bugs . . . 6

2.3.2. Frequency of upgrade bugs . . . 7

2.3.3. Frequency of input-related bugs . . . 8

2.4. Categories of Upgrade Bugs . . . 9

2.4.1. Categorization of environment-related upgrade bugs . . . 9

2.4.2. Frequency of input and/or environment-related upgrade bugs . . 11

2.5. Discussion . . . 12

2.6. OpenSSH upgrade bugs . . . 12

**3. Mojave: An Upgrade Recommendation System** . . . 15

3.1. Introduction . . . 15

3.2. Motivating Example . . . 16

3.3. Overview . . . 18
3.3.1. Learning phase . . . 18
3.3.2. Recommendation phase . . . 20
3.4. Design and Implementation . . . 21
3.4.1. Learning Phase . . . 21
3.4.2. Recommendation Phase . . . 28
3.4.3. Discussion . . . 29
3.5. Evaluation . . . 31
3.5.1. Methodology . . . 31
3.5.2. Results . . . 35

**4. Sahara: An Upgrade Debugging System** . . . 42
4.1. Introduction . . . 42
4.2. A Motivating Example . . . 43
4.3. Design and Implementation . . . 45
4.4. Discussion . . . 51
4.5. Evaluation . . . 54
4.5.1. Methodology . . . 54
4.5.2. Results . . . 56

**5. Related Work** . . . 65
5.1. Characterizing Upgrades . . . 65
5.2. Upgrade Deployment and Testing . . . 66
5.3. Recommendation Systems . . . 66
5.4. Automated Debugging . . . 66

**6. Conclusion** . . . 73

**References** . . . 75

# List of Tables

3.1. Experimental setup parameters. . . . . . . . . . . . . . . . . . . . . . . . . . . 32

3.2. Parsers. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

3.3. Mojave experiments. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

3.4. Results for the learning phase (Port = Port forwarding; X11 = X11 forwarding; Proxy = ProxyCommand; SSim = SSimilarity; FSim = FSimilarity). . . . . . 35

3.5. Recommendations for OpenSSH (three), SQLite (one) and uServer (one) bugs (Port = Port forwarding; X11 = X11 forwarding; Proxy = ProxyCommand). . 37

4.1. Sahara experiments. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

4.2. Results for three OpenSSH bugs (Port = Port forwarding; X11 = X11 forwarding; Proxy = ProxyCommand; dRs = `diff` Routines; SRs = Suspect Routines; DRs = Deviated Routines). . . . . . . . . . . . . . . . . . . . . . . . 57

4.3. Impact of number of profiles with failure-inducing settings (Port = Port Forwarding; X11 = X11 forwarding; Proxy = ProxyCommand; SRs = Suspect Routines; DRs = Deviated Routines). . . . . . . . . . . . . . . . . . . . . . . . 62

# List of Figures

2.1. Environment-related bugs in OpenSSH. . . . . . . . . . . . . . . . . . . . 7

2.2. Upgrade bugs in OpenSSH. . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.3. Input-related bugs in OpenSSH. . . . . . . . . . . . . . . . . . . . . . . . 9

2.4. Environment-related upgrade bugs in OpenSSH. . . . . . . . . . . . . . 10

2.5. Categories of environment-related upgrade bugs in OpenSSH. . . . . . . 10

2.6. Environment and input related upgrade bugs in OpenSSH. . . . . . . . 11

3.1. Learning phase in Mojave. . . . . . . . . . . . . . . . . . . . . . . . . . . 19

3.2. Recommendation phase in Mojave. . . . . . . . . . . . . . . . . . . . . . 21

3.3. Def-use chain, suspect variables and routines for the example. . . . . . . 25

3.4. Call sequences for the example program. . . . . . . . . . . . . . . . . . . 27

4.1. Overview of Sahara. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

4.2. Def-use chain, suspect variables and routines for Sahara's simple example. 48

# List of Code Listings

3.1. Example upgrade . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

4.1. Sahara: Example program. . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

4.2. Execution log of current version. . . . . . . . . . . . . . . . . . . . . . . . . 50

4.3. Execution log of upgraded version. . . . . . . . . . . . . . . . . . . . . . . . 51

# Chapter 1
# Introduction

Modern software systems are complex and comprise many interacting and dependent components. Upgrades for such systems are frequent, and may involve new and enhanced versions of some or all components, added or removed features, patches or bug fixes, or other software modifications. Unfortunately, the process of deploying and integrating these upgrades is labor-intensive and error-prone. For developers, it is infeasible (or maybe impossible) to foresee, much less test their software for every possible environment setting and input with which the user drives the software. As a result, many of the upgrades either crash the software or produce unwanted behavior for some users. We refer to these scenarios as upgrade failures. Upgrade failures can cause severe disruption, user dissatisfaction, and potential monetary loss.

A survey conducted by Crameri *et al.* [12] showed that 90% of system administrators perform upgrades at least once a month, and that 5 - 10% of these upgrades are problematic. Interestingly, they also found that the most common source of upgrade failures is the difference between the environment (i.e., version of operating system and libraries, configuration settings, environment variables, hardware, etc.) at the developer's site and the users' sites. To further study the prevalence of upgrade failures (among all the issues reported), and the reasons for these problems, we surveyed 96 bug reports spanning five versions of OpenSSH. The results show that a significant percentage of the bugs reported in OpenSSH are due to software upgrades. Furthermore, the survey confirms that most upgrade failures are caused by one or more aspects of the user's environment, and/or the user's input.

In an attempt to avoid installing buggy upgrades, developers typically deploy upgrades as packages to be handled by package-management systems. However, package-management systems only try to enforce that the right packages are in place; they provide no help with testing the upgrades for correct behavior at individual user sites. To make matters worse, when the upgrades misbehave, developers are given limited (and often unstructured) information to pinpoint and correct the problems. In some cases, the developers may also receive logs of failed executions and/or core dumps. Developers often undergo several exchanges with the users to gather all the pertinent information. Thereafter, the developers examine the information to locate the likely causes of the misbehavior. This process is long and tedious, as developers may have to consider large chunks of code to locate the root cause of the misbehavior, provide a (locally tested) fix, and restart the deployment process.

One approach that users take is to delay installing the upgrades until after a significant number of other users have installed and provided positive feedback for that upgrade. The survey [12] showed that 70% of the administrators refrain from installing the upgrade, regardless of their experience level. However, feedback from other users may be incomplete or not relevant if the user's environment and/or input is (significantly) different.

Obviously, neither deploying upgrades as packages nor delaying upgrades is enough to prevent upgrade failures. Furthermore, the user feedback is typically scattered across various mailing lists and discussion forums. As a result, the process of collecting and aggregating it is riddled with problems and inefficiencies.

In this dissertation, we argue that the developer and the users could collaborate to prevent many of the upgrade failures, and simplify the debugging of such failures. The developer can aggregate feedback from the (willing) users that have installed the upgrade, along with their environment settings and execution data, to predict success or failure for new users, and aid in debugging of the observed failures. To this end, we present two systems that simplify the management of software upgrades. Both systems rely on environment information and dynamic execution data collected from users who have previously upgraded the software (we refer to these users as "existing users"). First, we propose Mojave, a recommendation system for software upgrades that predicts the likelihood of upgrade failures for new users, given their environment settings and execution behavior with the current version of the program. Second, we propose Sahara, a system that simplifies the debugging of environment-related upgrade failures by pinpointing the subset of routines and variables that is most likely the source of misbehavior.

Mojave collects and aggregates success/failure feedback from (willing) existing users, along with their environment settings and execution behavior data from before the upgrade was installed (collectively called user "attributes"). It then uses machine learning, and static and dynamic source analyses to identify the attributes that are most likely related to the upgrade failures. Mojave then compares these suspect attributes to those of each new user to predict whether the upgrade would fail for him/her. Based on this prediction, Mojave recommends in favor or against the upgrade.

The developer may use the number of times Mojave recommends against an upgrade as an indication of the quality of the upgrade. This quality measure may be leveraged to prioritize the debugging of the most critical failures (to reduce the number of negative recommendations for future users). Furthermore, the developers can plan sophisticated upgrade testing plans that would include testing the program with the user attributes that are highly correlated to the observed failures.

In cases when the new user decides to upgrade (regardless of Mojave's recommendation) and it fails, and for the upgrade failures observed by the initial users, our second system, Sahara, comes into play. Sahara identifies the aspects of the environment that are most likely the culprits of the misbehavior, finds the subset of routines that relate directly or indirectly to those aspects, and selects an even smaller subset of routines to debug first. Sahara leverages feedback from a large number of user sites, machine learning, and static and dynamic source analyses to compute prime suspects (including the culprit routines).

Sahara provides valuable insights in the form of the suspect aspects of the user's environment, and a small set of routines that are highly correlated with the upgrade failures. This data can reduce the debugging time and effort significantly. In addition, the developers can gather this information over a period of time, and identify the environment settings that are the most (or least) frequently (and highly) correlated with the upgrade failures. Such knowledge can reveal the most critical (or bug-revealing) testing environments, and can be leveraged to perform more informed upgrade deployment and testing.

In summary, this thesis makes the following contributions:

- We characterize 96 bugs spanning 5 versions of OpenSSH.
- We propose Mojave, the first recommendation system for software upgrades.
- We propose Sahara, a system for simplifying upgrade debugging that is driven by user environments and includes a novel combination of techniques.
- We evaluate Mojave and Sahara with five upgrade failures from three widely used applications.

Based on our experience and results, we conclude that Mojave and Sahara can be extremely useful to software developers and users in practice. More fundamentally, we conclude that combining user feedback, machine learning, and static and dynamic analyses can achieve excellent results in preventing and debugging upgrade failures.

# Chapter 2
# Bug Characterization

## 2.1 Introduction

The surveys done by Beattie *et al.* [5], Crameri *et al.* [12], and the Secunia survey [48] revealed interesting results: (a) upgrade failures are frequent [5, 12]; (b) differences in vendor and user environments are a major source of upgrade failures [12]; and (c) the majority of administrators delay deploying upgrades to avoid the impact of buggy upgrades [12,48]. However, none of these works studied (a) multiple versions of a single software in detail; (b) the prevalence of upgrade problems among all the bugs reported for a software; and (c) the correlation between upgrade failures and the various aspects of user's attributes (environment and the input)

To bridge these gaps in the literature, we surveyed the problems reported in five versions of the `ssh` and `sshd` applications in the OpenSSH suite. OpenSSH is an open source implementation the SSH connectivity tools derived originally from `ssh 1.2.12`. OpenSSH encrypts all traffic (including passwords) to effectively eliminate eavesdropping, connection hijacking, and other attacks. Additionally, OpenSSH provides secure tunneling capabilities and several authentication methods, and supports all SSH protocol versions.

OpenSSH comprises many components: (1) *sshd*, the daemon that listens for connections coming from clients; (2) *ssh*, the client that logs and executes commands on a remote machine; (3) *scp*, the program to copy files between hosts; (4) *sftp*, an interactive file transfer program atop the SSH transport; and (5) utilities such as *ssh-add*, *ssh-agent*, *ssh-keysign*, *ssh-keyscan*, *ssh-keygen*, and *sftp-server*. In all, OpenSSH has around 400 distinct files and 50-70K lines of code (LOC).

We chose OpenSSH because it is widely deployed in diverse user environments. Its upgrades are fairly frequent, typically once every 3-6 months [41].

## 2.2 Methodology

We manually inspected the OpenSSH bugzilla to estimate the frequency and types of the bugs reported, the types of upgrade bugs, and the frequency of types of upgrade bugs. Our intention was to study around 100 bug reports spanning 5 versions of OpenSSH. The following set of five versions comprises 99 bug reports: 4.1p1, 4.2p1, 4.3p1, 4.3p2, and 4.5p1. However, we considered only 96 of those bug reports because two out of the 99 bugs were not valid (the users never followed up on those bugs and the developers could not reproduce the problems), and one bug was a duplicate. For every bug report, we determine if the bug is: (a) an upgrade bug (caused by a software upgrade), and if so, in which version it was introduced; (b) an environment-related bug; (c) an input-related bug (a specific set of inputs activates the bug); (d) a misconfiguration; or (e) a combination of the previous categories.

## 2.3 Survey Results

In this section, we present the results of the characterization of 96 OpenSSH bugs. In addition, we explore the categories of the upgrade related bugs in detail.

### 2.3.1 Frequency of environment-related bugs

We analyze the bug reports to ascertain if the bugs were caused by one or more aspects of the user's environment. Note that the user's environment comprises the version of operating system and libraries, configuration settings, environment variables, hardware, and so on. Figure 2.1 illustrates that $80-97\%$ of the bugs were environment related. The high number of environment-related bugs indicates that (a) many bugs are uncovered in the users' environment; and (b) the developer is unable to test the software in many diverse environments. This result confirms previous studies [12] in the context of a specific widely used application.

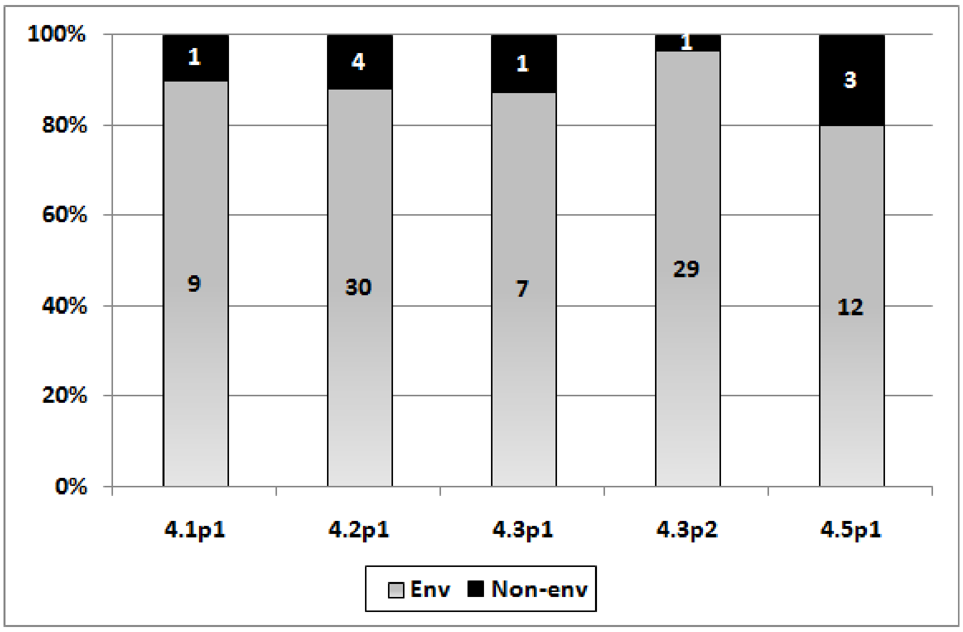


Figure 2.1: Environment-related bugs in OpenSSH.

Given such a high frequency of environment-related bugs, the developer would benefit from testing the software in diverse user environments during the software deployment cycle, and recording their feedback. In addition, identifying the aspects of the users' environment that are most frequently correlated with the reported bugs can provide valuable insights for testing of the future versions.

### 2.3.2 Frequency of upgrade bugs

For every bug, we determine the version in which it was discovered, and whether it resulted directly or indirectly from a source change due to an upgrade. If the bug was introduced as result of an upgrade, it is considered an upgrade bug, otherwise it is considered a bootstrapping bug. A bootstrapping bug is the one that has always been present in the source code, but not discovered until the version in which it was reported. Note that some of the upgrade bugs may have been introduced in the earlier versions, but were reported in a later version. This mismatch could occur because (1) many users do not update their software every time an upgrade is released; (2) the users did not exercise the buggy code; or (3) the bugs are non-deterministic.

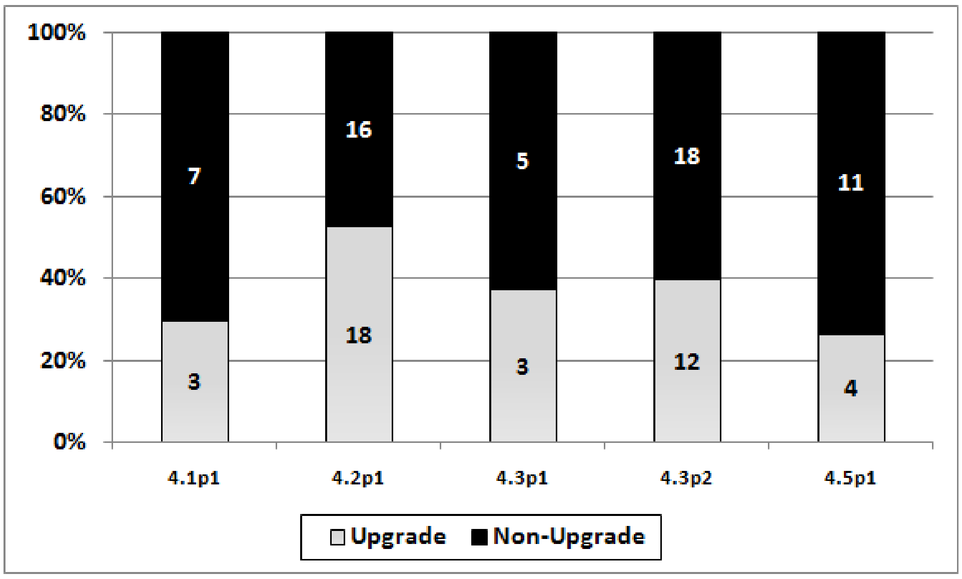


Figure 2.2: Upgrade bugs in OpenSSH.

Figure 2.2 shows that 27 – 53% of all the bugs reported for each version were upgrade failures. The large number of upgrade bugs indicates that: (a) many upgrades misbehave at user sites; and (b) the developers cannot test for all environment settings and/or inputs. Many of the upgrade bugs could be prevented if the user site testing were integrated in the software deployment cycle.

The presence of a large number of bootstrapping bugs even in the later versions of OpenSSH implies that, although the software is upgraded frequently to fix bugs, many latent bugs linger on in the software until much later in its life-cycle. Furthermore, as evident from the Figure 2.2, the total number of bugs (and the types of bugs) reported against a version does not depend on the number of revisions of the software.

### 2.3.3 Frequency of input-related bugs

We examined the reports to find out the inputs that activate the bug. A bug can be activated by multiple input strings and may be dependent on the user's environment settings. As shown in Figure 2.3, 70 – 100% of the reported bugs are triggered by specific input strings. Therefore, a dynamic component that can reflect the impact of users' inputs is crucial for understanding and dealing with upgrade problems.

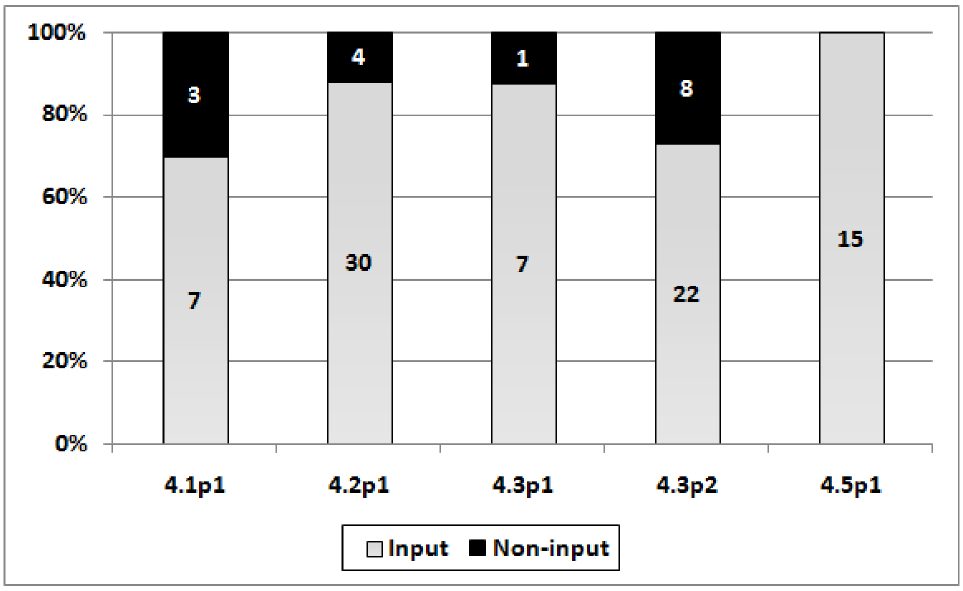


Figure 2.3: Input-related bugs in OpenSSH.

## 2.4 Categories of Upgrade Bugs

Now we focus solely on the upgrade bugs. We classify them as environment-related, input-related, or both. In addition, we categorize the environment-related upgrade bugs based on the specific aspects of the user's environment. A bug may belong to more than one category as these categories are not mutually exclusive.

### 2.4.1 Categorization of environment-related upgrade bugs

This category captures the upgrade failures that were caused by one or more aspects of the user's environment. As shown in Figure 2.4, the majority of the upgrade bugs are environment-related. Specifically, $77 - 100\%$ of the upgrade bugs are caused by one or more aspects of the user environment. Next, we breakdown the user environment into three categories: (a) application-specific environment (OpenSSH configuration settings), (b) system-environment (system libraries and their version data, shell environment, OS, etc.), and (c) hardware. Note that a bug can be caused by more than one aspect (and/or category) of the environment. Figure 2.5 plots the breakdown of the environment-related upgrade bugs by category. The majority of the environment-related upgrade bugs are caused by the application environment $(33-75\%)$

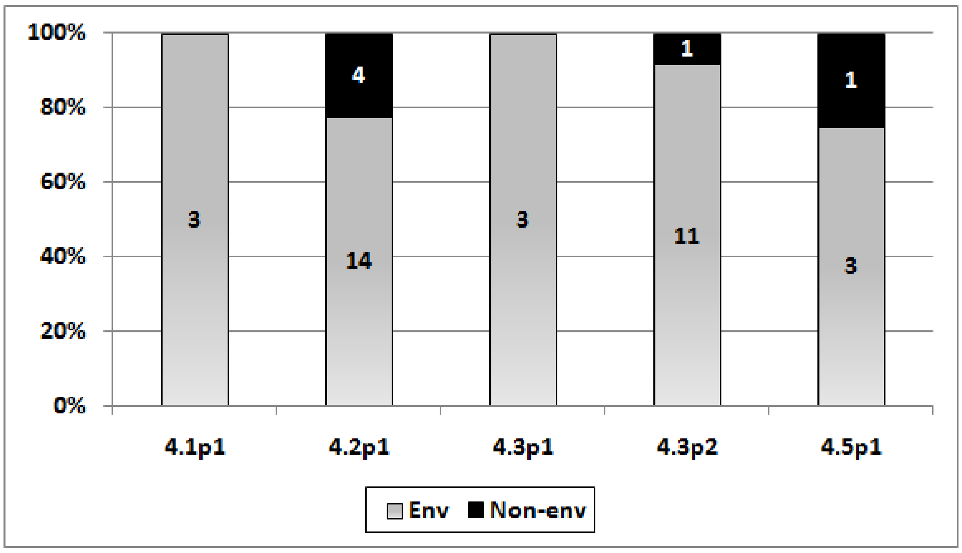


Figure 2.4: Environment-related upgrade bugs in OpenSSH.

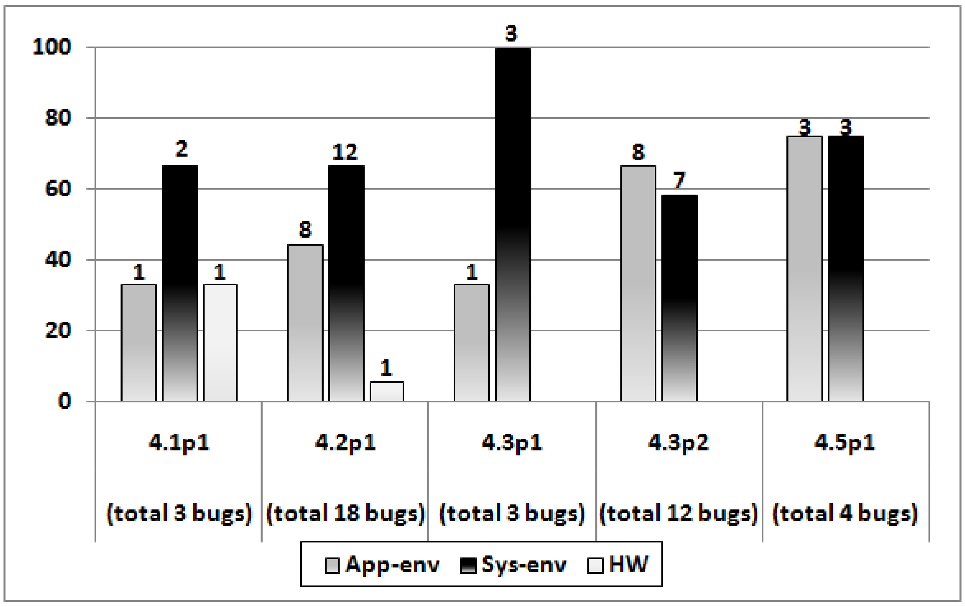


Figure 2.5: Categories of environment-related upgrade bugs in OpenSSH.

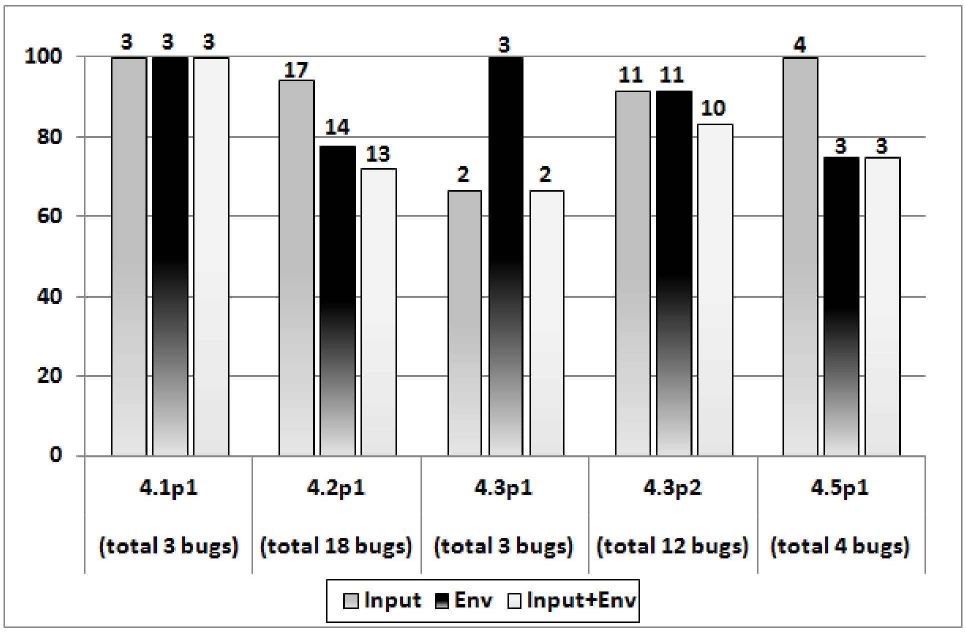


Figure 2.6: Environment and input related upgrade bugs in OpenSSH.

and/or system environment settings (66 – 100%); very few are caused by the hardware (0 – 33%).

Given such a high frequency of environment-related upgrade bugs, it is beneficial for the developer to (a) include the users in the upgrade testing cycle to enable testing in diverse user environments; and (b) collect their environment information and their feedback with (current and previous versions of) the software. The developer can analyze this user data to better allocate resources for upgrade deployment and testing, and debugging of upgrade failures. For instance, the developer could (1) test the new version of the software first at the user sites with environment settings that reveal the most bugs; or (2) choose to release the new version to only user sites where the upgrade can be installed with high confidence.

### 2.4.2 Frequency of input and/or environment-related upgrade bugs

This category captures the bugs that are caused by the user's environment and/or activated by his/her input for every version. Figure 2.6 demonstrates that a significant percentage of the upgrade bugs are both input- and environment-related (as described above). Therefore, capturing user's input (either directly or indirectly via execution traces, for instance) and environment settings, and testing the upgrade with those inputs and environment settings is imperative to deploy high quality upgrades. In addition, classifying users based on their inputs and/or environment settings can help predict upgrade failures for similar users.

## 2.5 Discussion

We did not strive to perform a comprehensive, statistically rigorous survey of all the bug reports in the field. Our main goals were to motivate Mojave and Sahara, and sample the data on real upgrade problems and classify them based on the causes of those failures. These restricted goals allowed us to focus on only OpenSSH bugs. Undoubtedly, this survey is narrower in its focus than other works in the literature [5, 12]. Nevertheless, our survey looks at upgrade problems in depth to determine the aspects of user attributes that cause them, and provides useful information about the ubiquity of upgrade problems and their various categories.

In summary, our analysis establishes that the significant number of OpenSSH bugs are actuated by upgrades, and are caused by the user's input and environment. In addition, the same trend is also observed for the upgrade problems: a majority of the upgrade problems were caused by the environment settings at the user sites and/or their input. Therefore, leveraging environment and execution (input) data from users can help prevent most of the upgrade failures, and expedite the debugging of those failures.

Next, we describe the three OpenSSH upgrade bugs that we use to evaluate Mojave and Sahara.

## 2.6 OpenSSH upgrade bugs

**Port forwarding bug.** Port forwarding is commonly used to create an SSH tunnel. To setup the tunnel, one forwards a specified local port to a port on the remote machine. SSH tunnels provide a means to bypass firewalls, so long as the site allows outgoing connections. The bug [8] was a regression bug in OpenSSH version 4.7. When using SSH port forwarding for large transfers, the transfer aborts. Some users observed the following buffer error:

```
buffer_get_string_ret: bad string length 557056
buffer_get_string: buffer error
```

These transfers executed successfully until version 4.6, but the behavior changed after upgrading to version 4.7. The failure was observed at a small subset of user sites. The abort was not reproducible at the developer site, so the developer needed volunteer users to reproduce the bug and test its fix. A correct and complete fix was submitted and tested by the users on the second attempt after almost three months from the time it was submitted [8].

The failure was caused by the following issues: (a) the users had enabled port forwarding in the *ssh* configuration file; (b) change in default window size from 128KB to 2MB in the *ssh* client code in version 4.7; (c) port forwarding code advertising the default window size as the default packet size; and (d) the maximum packet size set to 256KB in *sshd*. Given these characteristics, when users issued large transfers through the *ssh* tunnel, some of the packets had size larger than the daemon's maximum, resulting in the buffer error after the upgrade. The port forwarding code using the default window size as the default packet size was not an issue before the upgrade, as the size was always below the maximum.

**X11 forwarding bug.** This bug [7] manifested when users upgraded to OpenSSH version 4.2p1 from 4.1p1 and tried to start X11 forwarding. The failure was observed at the user sites that had X11 forwarding support enabled and the command was executed in the background. Users observed the following error:

```
xterm Xt error: Can't open display: localhost:10.0
```

In version 4.2p1, developers modified the X11 forwarding code to fill a number of X11 channel leaks, including destroying the X11 sessions whose session has ended. As a result, when the X11 forwarding process is started in the background, the child (and the channel) starting it would exit immediately. It took the developers more than two weeks to fix this bug [7].

**ProxyCommand bug.** The ProxyCommand option specifies the command that will be used by the SSH client to connect to the remote server. The bug [43] was a regression in OpenSSH version 4.9; ssh with ProxyCommand would fail for users with a `"No such file"` error.

Until version 4.7, ProxyCommand would use `/bin/sh` to execute the command. However, in version 4.9, the code changed to use the `$SHELL` environment variable, causing the command to fail at user sites where `$SHELL` was set to an empty string. The developers fixed this bug in one week, after one user had already done a large amount of debugging [43].

Chapter 3

# Mojave: An Upgrade Recommendation System

## 3.1 Introduction

It is difficult for the developers to deploy upgrades that will integrate properly into the users' systems and that will behave as users expect. The developers simply cannot anticipate and test their upgrades for all the applications and configurations that may be affected by the upgrades at the users' sites. As a result, enterprise users often wait for the upgrades to mature before applying them to a large number of machines. Individual users often postpone their upgrades arbitrarily or until they find enough positive feedback about the upgrade in public forums.

Obviously, neither delaying upgrades nor seeking feedback from other users is enough to prevent upgrade failures. Instead, we argue that the developer and the users of the software can collaborate to achieve this goal. The developer can aggregate data from the "existing users" (users who have already installed the upgrade) and use it to predict success or failure for "new users" who intend to upgrade their software.

Along these lines, we propose Mojave, the first recommendation system for software upgrades. Mojave's design is based on three observations: (1) the upgrade failures are most likely caused by the particular characteristics of the corresponding users' environments or inputs; (2) new users that have characteristics similar to those of the existing users where the upgrade has failed are likely to experience similar failures; and (3) if the behavior of the software at two users' sites was similar in the past (execution behavior prior to the upgrade), then it is likely to behave similarly in the future (execution behavior after the upgrade). This latter observation is the basis for many recommendation systems based on collaborative filtering [22, 47, 49].

Given these observations, Mojave gather the following information from (willing) existing users: (1) success/failure feedback, (2) their environment settings, and (3) their execution behavior data from before the upgrade was installed. Mojave aggregates this data with machine learning, and static and dynamic source analyses to identify the attributes that are most likely related to the upgrade failures. Mojave then compares the attributes of a new user with these suspect attributes to ascertain whether the upgrade would fail for him/her. Based on this prognosis, Mojave recommends in favor or against the upgrade.

A recommendation against the upgrade means that the user should wait until the developer has debugged the upgrade for the existing users for which it has already failed. However, the new user may decide to upgrade anyway.

We evaluate Mojave on three real upgrade problems with the widely used OpenSSH suite, one synthetic problem in the SQLite database engine, and one synthetic problem with the uServer Web server. The results demonstrate that Mojave produces accurate recommendations to the new users. The exact number of correct recommendations depends on the characteristics of the failure and the user feedback. Across all our experiments, Mojave accurately predicts the upgrade failure for $93-100\%$ of the users. Given its accuracy, we expect that Mojave would be able to prevent most upgrade failures in the field.

## 3.2 Motivating Example

Let us look at a simple example in Figure 3.1. The example reads input strings (lines 16-18) one at a time. If one of the strings is $Option1$, it calls `do_option` (lines 18-19) to process them (lines 10-12). If one of the strings is *Proxycmd*, it assigns `PSHELL` as the shell, computes its length, and calls `checklength` (line 21-24). `checklength` checks if the length of the string is less than or equal to 9, returns the length of the string if it is, $-1$ otherwise (lines 4-9).

Now let us assume that the upgrade replaces line 22 with the following three lines to get the value of `$SHELL` from the user's environment.

```
#define PSHELL ”/bin/sh”
int env2 = 0;

int checklength(int len) {
  if (len <= 9)
    return len;
  else
    return -1;
}
void do_option(int x) {
  printf(”\nGot option %d”,x);
}
int main(int argc, char *argv[]) {
  int retval1 = 0, i = 1;
  char shell[80];
  if (argc >= 2) {
    while (argv[i] != NULL) {
      if (strcmp(argv[i],”Option1”) == 0) {
        do_option(1);
      }//end strcmp
      if (strcmp(argv[i],”Proxycmd”) == 0) {
          strcpy(shell,PSHELL);
        env2 = strlen(shell);
        retval1 = checklength(env2);
        if (retval1 > 0)
          printf(”\nSuccess:proxycmd”);
        else
          printf(”\nOops:checklength failed”);
      }//end strcmp
      i++;
    }//end while
  }//end argc
}//end main
```

Listing 3.1: Example upgrade

```
strcpy(shell, getenv("SHELL"));
if (shell == NULL)
   strcpy (shell,PSHELL);
```

The upgrade will fail at the user sites where (1) *Proxycmd* is passed as input, and (2) the `$SHELL` variable is a string of length 0. Note that a `NULL` string is different from a string of length 0. Specifically, the upgrade will fail when `checklength` returns 0, because the length of the shell variable is 0. However, the program ran successfully at these sites before the upgrade, because it was not dependent on the user's setting of `$SHELL`. This upgrade failure is similar to the ProxyCommand bug [43] that we detail in Section 2.6.

Despite the fact that the two versions are input-compatible, the execution behavior changes with the upgrade, both in terms of the path (call sequence) executed, and the output produced. This was not the intended behavior of the upgrade. Therefore, the key to preventing this failure for future users is to check if they have the failure-inducing environment (`$SHELL` set to an empty string), and if their execution path with the current version of the software has the failure-relating routine call, `checklength`.

Mojave avoids asking the users for their input because of privacy concerns. Instead, Mojave provides the user with the instrumented version of the software, which automatically logs the sequence of calls executed at the user site, excluding any confidential data (e.g. routine arguments and return values). The call sequence does not uniquely correspond to the user input, however it represents the program's execution behavior at the user's site, and can be leveraged to predict its behavior after the upgrade.

## 3.3 Overview

Mojave comprises two phases: a) learning phase, and b) recommendation phase. In the learning phase, Mojave deploys and tests the upgrade at initial users' sites, collects their feedback, and uses the feedback to learn a prediction model for the upgrade. Mojave continues in the learning phase till the prediction accuracy on the training set becomes high. Mojave then moves to the recommendation phase, when the new users can request a recommendation by providing their environment and execution behavior data before deciding to install the upgrade.

### 3.3.1 Learning phase

Figure 3.1 details the steps that Mojave takes in the learning phase. First, Mojave deploys the upgrade to an initial set of users (step 0). Mojave then collects user input, environment, and call sequence data for the *current (pre-upgrade) version* of the software (step 1). The user input includes any command line arguments and data manually entered as input. The environment information includes the version of the operating system, the version of the libraries, the configuration settings, the name and

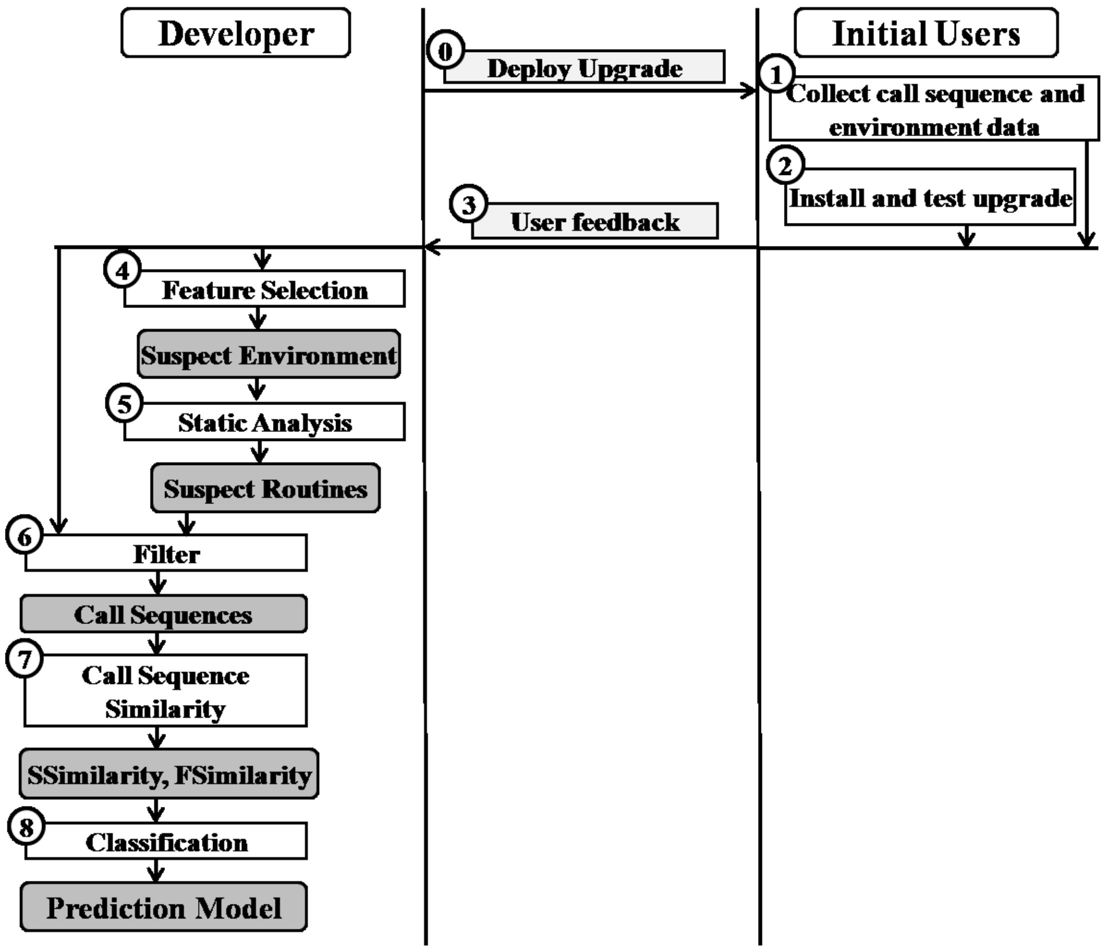


Figure 3.1: Learning phase in Mojave.

version of the other software packages installed, and a description of the hardware [12]. A call sequence comprises the routines executed during one run of the software. Mojave may collect multiple call sequences from each user.

After these pre-upgrade runs, Mojave installs the upgrade and tests it with the previously saved inputs (step 2). At the end of each test run, Mojave asks the user for a success/failure flag. When the user provides it, Mojave sends this information, along with the environment and call sequence data, back to the developer's site (step 3).

Mojave collects these data from all users that are willing to participate. Now suppose that the upgrade misbehaved at one user site at least. With the users' environment and upgrade success/failure information, Mojave runs a machine learning algorithm to determine if the misbehavior is likely to be caused by any aspects of the environment (step 4). Next, using def-use static analysis, Mojave isolates the variables in the pre-upgrade code that derive directly or indirectly from those aspects; the routines that use these variables are considered suspect (step 5).

Mojave then filters the call sequence(s) from the initial users into sequence(s) connecting only the suspect routines (step 6). Next, Mojave measures the similarity between the call sequence(s) from a user's site and other users where the upgrade succeeded and failed respectively (step 7). With the two similarity measures, and the environment and success/failure data from all users, Mojave runs a classification algorithm to build a prediction model (step 8).

### 3.3.2 Recommendation phase

Mojave stays in the learning phase till the prediction accuracy on the training set becomes high. When this is achieved, Mojave moves to the recommendation phase. Figure 3.2 illustrates this phase.

When a new user arrives to download the upgrade, Mojave first collects the user's environment and call sequence information (step 9), and transfers this data to the developer site (step 10). If the upgrade is likely to have environment-related bugs, Mojave filters the call sequence(s) of the new user into sequence(s) containing only the suspect routines (step 11). Thereafter, Mojave computes the success and failure similarity measures for each sequence of the new user, combines them with the new user's environment, and passes the combined attributes to the prediction model (step 12). The prediction model outputs if the upgrade is likely to succeed or fail at the new user's site, and accordingly recommends for or against the upgrade (steps 13 and 14). Note that Mojave does filtering and similarity computation using existing user data for the same version of the software currently installed at the new user's site.

Next we detail the implementation of these steps.

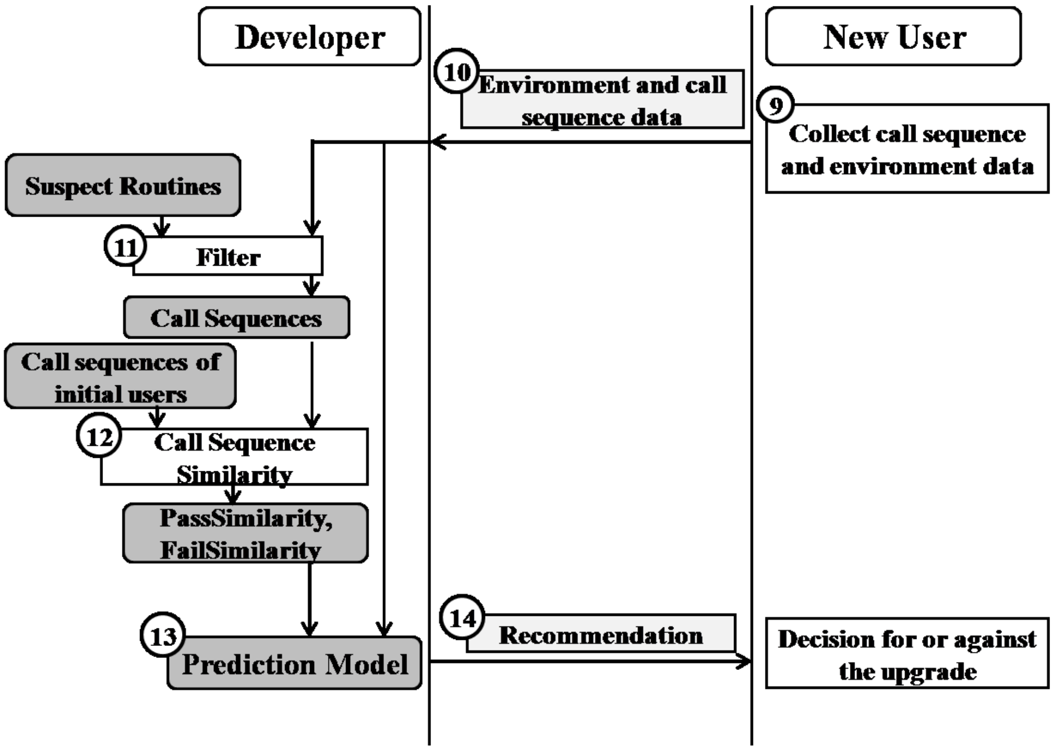

Developer
New User
10 Environment and call sequence data
9 Collect call sequence and environment data
Suspect Routines
11 Filter
Call Sequences
Call sequences of initial users
12 Call Sequence Similarity
PassSimilarity, FailSimilarity
13 Prediction Model
14 Recommendation
Decision for or against the upgrade

Figure 3.2: Recommendation phase in Mojave.

## 3.4 Design and Implementation

### 3.4.1 Learning Phase

**Upgrade deployment, tracing, and user feedback (steps 0-3).** Upgrade deployment in Mojave is trivial. The upgraded code is available via a Web interface and can be downloaded as a package/patch by any user that wants it. Mojave uses the Mirage tracing infrastructure, which has been described in detail in [12]. For this reason, next we only describe the most important aspects of it. The infrastructure identifies the "environmental resources" an application depends on and then fingerprints (i.e., derives a compact representation for) them.

The infrastructure creates a log of all the external resources accessed by an application by intercepting process creation, read or write, file descriptor-related and socket-related system calls. For environment variables, it intercepts the calls to the *getenv()* function in libc. The log may include data files, in addition to environmental resources. To separate them out, Mojave uses a four-part heuristic to identify the environment resources from multiple runs of the application. The heuristic identifies as environmental resources: (1) all files accessed in the longest common prefix of the sequence of files accessed in the logs; (2) all files accessed read-only in all logs; (3) all files of certain types (such as libraries) accessed in any single log; and (4) all files named in the package of the application to be upgraded. This heuristic allows Mojave to exclude unimportant files, such as temporary and log files, that are written but never read by the application. To complement the heuristic, Mojave also includes an API that allows the developer to include or exclude files or directories. Mojave also collects information about the hardware and software installed, such as type and amount of memory, CPU data, the types and number of devices present, and the list of kernel symbols and modules.

Mojave creates a concise representation (fingerprint) for each environmental resource. Depending on the resource type, a different fingerprint is generated. First, Mojave provides parsers that produce the fingerprint for common types such as libraries and executables. A parser knows how to extract the relevant information from a file based on its type. Second, the developer may provide parsers for certain application-specific resources, such as configuration files. Third, if there are no parsers for a resource, the fingerprint is a sequence of hashes of chunks of the file that are content-delineated using Rabin fingerprinting [46]. In practice, we expect most resources to be handled by parsers, so resorting to Rabin fingerprinting should be the exception.

In each fingerprint, the name of the resource serves as a key and the hash of its contents as the value. The parsers for the most common resource types produce fingerprints in the following formats:

- Environment variables: Name:HASH
- Libraries: Name:HASH+Version
- Configuration files: Filename.KEY:HASH
- Binary files: Filename:CHUNK_HASH

The content-based fingerprints are of the form: Filename:CHUNK_HASH. These fingerprints are more coarse grained than what is possible with parsers, since a parser can choose the granularity at which the fingerprint for an environment resource is produced. For instance, the granularity at which binary files are fingerprinted is typically coarser than that for configuration files. We use SHA-1 to compute fingerprints of the resources.

For the users that choose to participate, Mojave sends the tracing infrastructure and the parsers to their sites. During the first several executions of the upgraded software (the number of executions can be defined by the developer), Mojave collects the environment resource information and produces the respective fingerprints. After each of these executions, Mojave also queries the user about whether the upgrade has succeeded or failed. We ask the user to provide this success/failure flag, because it may be difficult to determine failure in some cases. For example, a software misbehavior is considered a failure, even if it does not cause a crash or any other OS-visible event. In addition, the upgrade may cause another software to misbehave [12].

In addition to environment data, Mojave collects the execution call sequences for the software before the upgrade. Mojave uses the *C Intermediate Language* (CIL) [38] to automatically instrument the application. The instrumentation is introduced by a new CIL module, *call-logger*. The call-logger module inserts calls to our runtime library that logs names of all the routines executed in a particular invocation of the software. In case multiple processes are forked during an instance, it logs the call sequence for each process individually. Mojave drives the instrumented version of the software with the previously saved input to generate the call sequences.

The developer can configure Mojave to collect call sequences corresponding to multiple executions of the (pre-upgrade) software at each user site. For clarity and without loss of generality, the remainder of this dissertation assumes that Mojave collects call sequence data corresponding to a single execution at each user site. The extension to multiple executions per site is straightforward.

After the user provides each success/failure flag, Mojave obscures and then transfers the collected environment fingerprints and call sequence data to the developer's site, along with the flag. The call sequences (excluding any confidential data such as routine arguments and return data) are a proxy for the real environment settings and inputs, which Mojave does not transfer to the developer because of privacy concerns. These data represent the profile of the user site. User profiles from all sites serve as the input to the other steps.

**Call sequence filtering for environment-related failures (steps 4-6).** Based on the environment information collected from the user sites, this step ascertains if the upgrade failure is environment-related. Mojave runs feature selection on the environment data and the success/failure flags to select environment resources (called features) with the strongest correlation to the observed upgrade failures. The fingerprints are never "unhashed" during feature selection (or after it); it is enough for Mojave to know how many different fingerprints there are for each feature.

Mojave uses the decision tree algorithm with feature ranking from the WEKA tool [32] for selection. The algorithm builds a decision tree by first selecting a feature to place at the root node, and creating a tree branch for each possible value of the feature. This splits up the dataset into subsets, one for each value of the feature. The choice of the root feature is based on Gain Ratio [45], a measure of a feature's ability to create subsets with homogeneous classes. In Mojave, there are only two classes: success or failure. The Gain Ratio is higher for the features that create subsets with mostly success or mostly failure user profiles.

For instance, in the example of Listing 3.1, the root feature would be the SHELL environment variable. The subsets that include SHELL strings of length greater than 0 characters are successes, whereas those that have strings of length 0 are failures.

After selecting the root feature, the process is repeated recursively for each branch, using only those profiles that actually reach the branch. When all the profiles at a node have the same classification, the algorithm has completed that part of the tree. The output of the algorithm is a set of features, their Gain Ratios, and their ranks.

To validate the feature selection, Mojave uses 10-fold cross-validation [25] to compute the standard deviation of the ranks of each feature. When the standard deviations of the top-ranked features are high, the reason for the failures is unlikely to

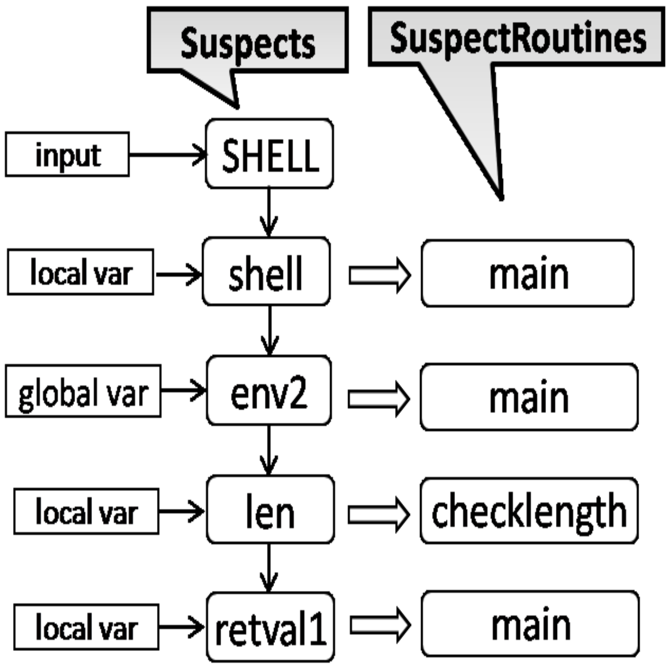


Figure 3.3: Def-use chain, suspect variables and routines for the example.

be the environment. In that case, Mojave skips to the call sequence similarity step.

When this condition is not met, Mojave considers all the features that have Gain Ratios within 30% of the highest ranked feature as *Suspect Environment Resources (SERs).* These SERs serve as input to the static analysis.

Mojave performs static analysis using CIL [38]. Specifically, it implements two CIL modules, the *call-graph* module and the *def-use* module. As the name suggests, the call-graph module computes a whole-program static call graph by traversing all the source files, a routine at a time. Every node in the call graph is a routine, and its children nodes are the routines it calls. The root of the call graph is always the `main()` routine.

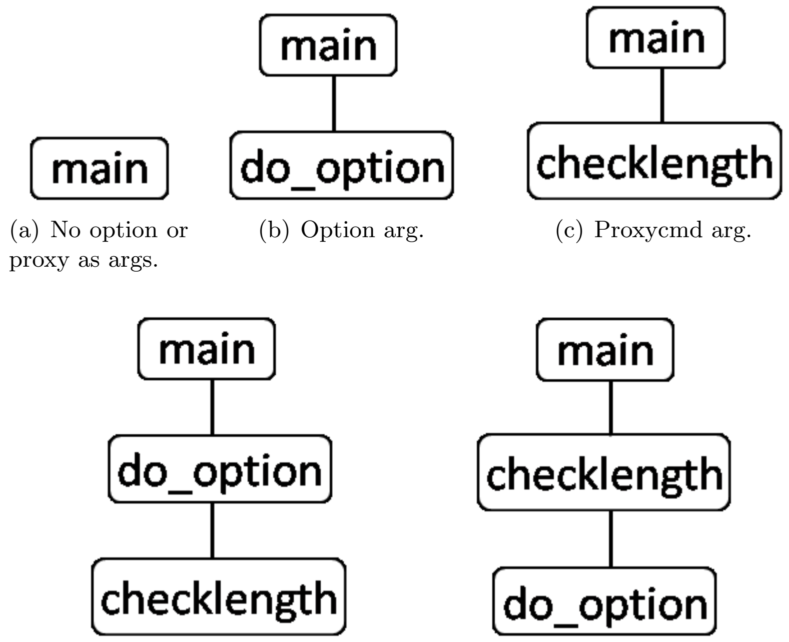
The def-use module creates def-use chains [1] for each SER. A def-use chain links all the variables that derive directly or indirectly from one SER. Each array is handled as a single variable, whereas struct and union fields are handled separately. Figure 3.3 shows the def-use chain (thin arrows) for our example program.

Since *SuspectRoutines* is the set of routines that are highly correlated with the failure, Mojave filters out the call sequence data to comprise only the sequence connecting the suspect routines (SCSR). Specifically, it removes all the routines that are not suspect from the execution call sequence, resulting in shorter sequences, and potentially faster similarity computation (next step). This step updates the call sequence data for all the users.

**Call sequence similarity (step 7).** In this step, Mojave determines how similar a (pre-upgrade) execution at a user site is (in terms of its call sequence) to other users' (pre-upgrade) executions where the upgrade succeeded or failed. Mojave measures similarity as the length of the longest common subsequence (LCS); the longer the LCS, the greater the similarity. Mojave computes the pairwise length of the LCS as a percentage, between call sequences for every existing user. For each user and sequence, Mojave then computes the $90^{th}$ percentile length of the LCS with the sequences of the users where the upgrade has failed (*FSimilarity*), and with those where the upgrade has succeeded (*SSimilarity*).

Figure 3.4 illustrates the possible call sequences for the example in Figure 3.1. Figure 3.4(a) shows the call sequence when the program is run without the input arguments “Option1” and “ProxyCommand”. Figures 3.4(b) and 3.4(c) depict the call sequence when the program is executed with “Option1” or “ProxyCommand” arguments, but not both. Figures 3.4(d) and 3.4(e) exhibit the call sequence when the program is executed with both arguments.

The LCS between the call sequences in Figure 3.4(a) and Figure 3.4(b), and between Figure 3.4(a) and 3.4(c) is the `main` routine. Since the length of the longer of the two sequences is 2 routines, the length of the LCS as a percentage is 50% for both these cases. Similarly, the length of the LCS as a percentage between Figure 3.4(a) and Figure 3.4(d), and between Figure 3.4(a) and 3.4(e) is 33% (the LCS contains only the `main` routine and the length of the longer sequence is 3 routines). Note that two users may have the exact same call sequence, in which case the length of the LCS as a percentage for those users would be 100%. The two similarity measures, *FSimilarity* and *SSimilarity*, range from $33-100\%$ for the example depending on the total number



(a) No option or proxy as args.

(b) Option arg.

(c) Proxycmd arg.

(d) Option and Proxycmd args.

(e) Proxycmd and Option args.

Figure 3.4: Call sequences for the example program.

of initial users, the number of users where the upgrade passed or failed, and their respective call sequences.

Mojave computes the two similarity measures for every user and sequence, and adds them as attributes to the user's profile. The updated profiles comprising environment data, call sequences, similarity measures, and upgrade success/failure labels form the training set for the classification step.

**Classification (step 8).** The classification step takes the user profiles and attempts to learn a binary classifier that gives good predictions on the test set. Specifically, Mojave uses the *Logistic Regression* [26] classification algorithm. Logistic Regression is a method of learning a binary classifier where the output function is assumed to be logistic. The logistic function is a continuous S-shaped curve approaching 0 on one end, and 1 on the other. The output can be interpreted as a probability that the data point falls within class 0 or 1. Quantizing the logistic function output then gives us a binary classifier: if the output is greater than 0.5, then the data point is classified as class 1 (failure), otherwise it falls in class 0 (success). The algorithm learns the relationship between all attributes (except for call sequences) and the binary class variable. It outputs a model that predicts whether an upgrade will succeed or fail for a new user, given his/her attributes. The model is a linear equation, where each term is the multiplication of an attribute rule (testing whether the attribute has a specific value) and a weight assigned to the rule. The following equation shows the model for the example in Figure 3.1:

$$p(fail) = a_0 + a_1 * (SHELL = HashOfOffendingShellName)$$

where $a_i$ are the weights that are learned from the training set, and $HashOfOffendingShellName$ is the hash of the failure-inducing shell name (empty string). If the value of $p(fail)$ is greater than 0.5, then the predicted class is *fail*, otherwise it is *success*. In some cases, the model may include separate equations for the two classes; the predicted class is the one with the higher of the two probabilities.

### 3.4.2 Recommendation Phase

**User feedback (steps 9-10).** In this step, Mojave collects the fingerprints of the new user's environment settings, and the call sequence(s) from the current version of the software. Mojave collects these data using the tracing infrastructure described above. Mojave then obscures and transfers the data back to the developer.

**Filtering for environment-related failures (step 11).** In this step, Mojave filters the new user's call sequence data with *SuspectRoutines* (computed in the learning phase) to contain only the SCSR, in cases when the existing users observed failures that are likely environment-related. The SCSR is then passed on to the call sequence similarity step.

**Call sequence similarity (step 12).** In this step, Mojave quantifies the similarity of the call sequence(s) from the current version of the software at the new user's site with the call sequences from the same version at the existing users' sites where the upgrade has succeeded or failed. Specifically, Mojave (a) computes the pairwise length of the

LCS of each user's sequence (or the SCSR if the failure is environment-related) with other users where the upgrade succeeded and failed, respectively; (b) takes the $90^{th}$ percentile length of the LCS to compute the two similarity measures, *SSimilarity* and *FSimilarity*, for each sequence of the new user; and (c) updates the user's profile with the similarity measures for each sequence. This step is similar to that performed in the learning phase to compute similarity between initial users.

Note that a new user may have skipped the most recent upgrades of the software. This does not pose a problem, since Mojave compares the new user's profile only to those of existing users who ran the same version of the software *and* have installed the current upgrade.

**Recommendation (steps 13-14).** Mojave inputs the user's updated profile to the prediction model (built in the learning phase) to compute the probability that the new user belongs to the class 0 (success) or 1 (failure). The predicted class for the new user is the one that has the highest probability. If the predicted class for the new user is success, Mojave recommends the upgrade to the user. Otherwise, Mojave recommends the user to not upgrade the software.

### 3.4.3 Discussion

**Upgrades that change the environment and/or call sequence.** Recall that Mojave uses environment and call sequence data about a pre-upgrade version of the software to predict its post-upgrade behavior for other users. Mojave works well even for upgrades that change the environment and the call sequence, because it also associates the post-upgrade success/failure flags from some users to their respective pre-upgrade data. With this post-upgrade information, Mojave can learn the pre-upgrade behaviors that will likely lead to success/failure for new users.

**Multiple bugs in an upgrade.** Most components of Mojave are unaffected by the presence of multiple bugs. However, multiple bugs may negatively affect the feature selection, similarity computation, and classification steps, when the bugs are caused by different environmental resources. In these cases, the feature selection might mis-rank the features that are related to the less frequent bugs, such that they are not selected as SERs. This could cause the static analysis to miss some suspect routines, causing some relevant calls to be filtered out of the sequences from the user sites exposed to the less frequent bugs. This in turn could make the similarity computation for those sites less accurate, causing the prediction model to become inaccurate. This inaccuracy would cause Mojave to stay in the learning phase longer, until more data could be collected on those less frequent bugs.

To reduce this delay in entering the recommendation phase, Mojave can be combined with systems that classify bugs into buckets, each of which comprising a single bug (e.g., [13]). In such cases, Mojave would compute a prediction model for each bucket. In the recommendation phase, Mojave could calculate the failure likelihood at the new user's site for each of the buckets, and provide a recommendation based on some aggregation of these likelihoods.

**Limitations of the current implementation.** Mojave limits the user information transferred to the developer's site to the resource fingerprints and call sequence data. In our current implementation, these data are transferred in hashed form (SHA-1), which does not provide foolproof privacy guarantees. However, Mojave can easily use more sophisticated schemes for these transfers. For long-running software (e.g., servers), the network bandwidth required by the call sequence transfers may be significant for the initial users (in the learning phase), since these sequences have to be transferred in their entirety. New users may transfer shortened sequences (filtered based on the suspect routines), but even those can be long. This sequence length problem can be mitigated by using sampling techniques, as in [2].

Mojave employs feature selection and static analysis to narrow the set of routines that are highly correlated with failures. However, under certain conditions, these techniques may be unable to do so. In the worst case, all routines may be affected by the SERs, making static analysis ineffective. Fortunately, these worst-case scenarios are extremely unlikely in a single upgrade. Furthermore, our results with the uServer bug show (Section 3.5) that Mojave can provide accurate recommendations even when these techniques are absent or not applicable.

Finally, Mojave currently computes the LCS between every pair of (possibly shortened) call sequences for the same version of the software in its database. We do not consider this high computational complexity a problem for large software vendors, since they can dedicate massive resources to these computations. For smaller vendors or open-source developers, this complexity can be a problem. Possible optimizations would be (1) using a subset of call sequences as representatives for the others, and/or (2) using an approach to similarity that does not involve computing the LCS. We leave these optimizations as future work.

## 3.5 Evaluation

In this section, we describe our methodology and evaluate Mojave with three real upgrade bugs in OpenSSH, a synthetic bug in SQLite, and a synthetic bug in uServer.

We describe OpenSSH in Section 2.1. SQLite is the most widely deployed SQL database [50]. It implements a serverless, transactional SQL engine. SQLite has 67K LOC spread across 4 files. uServer [10] is an open-source, event-driven Web server sometimes used for performance studies. It has 37K LOC spread across 161 files.

### 3.5.1 Methodology

We describe the OpenSSH bugs we study in Section 2.6. Next, we describe the bugs we introduced in SQLite and uServer.

**SQLite and uServer bugs.** To demonstrate Mojave's generality, we synthetically created one buggy upgrade for SQLite version 3.6.14.2 and one for uServer version 0.6.0. Note that these two bugs are trivial, however, our goal is simply to demonstrate that our systems works without modification for a variety of applications and it can also help prevent bugs that are not environment related.

Before the upgrade of SQLite, the option *echo on* caused its shell to output each command before executing it. After our synthetic upgrade, it does not output the command when executing in *interactive mode*. The bug we inject into the upgrade of

| Parameter Name | Default Value |
|---|---|
| Config files | 8 |
| Files with failure-inducing settings | 3 |
| Total user profiles | 87 |
| Failed user profiles | 20 |
| Total inputs | 8 |
| Failure-activating inputs | 3 |
| Feature selection threshold | 30% |

Table 3.1: Experimental setup parameters.

| Parser Name | Description |
|---|---|
| CHUNKS | Chunks and fingerprints a binary file into 1KB chunks |
| CHUNKS2 | Chunks and fingerprints a file into variable sized chunks |
| KEYVAL | Chunks and fingerprints a key-value pair file |
| LIBS4 | Chunks and fingerprints a library and all its dependencies |
| LINES.c | Fingerprints a file line-by-line |
| SSHD | Application-specific parser to fingerprint the sshd_config file |
| SSH | Application-specific parser to fingerprint the ssh_config file |

Table 3.2: Parsers.

uServer is *not* environment-related. The bug is a typo in the function that parses user input causing dropped `POST` requests and occasional crashes.

**Upgrade deployment and user data collection.** To simulate a real-world deployment of a software upgrade to a large number of users with varied environment settings, we collected environment data from 87 machines at our site across two clusters. The settings of the machines within a cluster are similar, but they are different across clusters. Table 3.1 lists the default values of the parameters in our experimental setup.

We used the methodology described in Section 3.4 to (1) automatically generate instrumented versions of OpenSSH, SQLite, and uServer; (2) identify their environmental resources; and (3) collect call sequence data and compute success and failure similarity measures. While Mojave collects the call sequence data and identifies the environmental resources, the software takes longer to execute. Specifically, for the five bugs we analyzed, the software ran $1.5X - 3X$ slower than when the data is not being collected. We ran all the experiments on 2.80GHz Intel Pentium 4 machines with 512MB RAM and the Ubuntu 8.04.4 Linux distribution.

Table 3.2 lists the parsers used to parse and fingerprint these environmental resources. CHUNKS and CHUNKS2 chunk and fingerprint the binary files, such as the kernel symbols; KEYVAL parses and chunks any file in the *key-delimiter-value* format, such as shell environment or cpu data; LIBS chunks and fingerprints all the libraries; LINES.c parses and fingerprints a file one line at a time, such as the file containing the list of kernel modules; and SSH and SSHD are application-specific parsers to parse and fingerprint the *ssh_config* and *sshd_config* configuration files, respectively. It took us only 8 person-hours to implement these parsers. SQLite and uServer did not require any application-specific parsers.

In addition, we downloaded eight different complete OpenSSH configuration files from the Web, and generated eight synthetic configurations for SQLite and uServer. For each of the bugs, we modify three of these files to include the settings that activate the bug. Furthermore, we use eight inputs, three of them would trigger the bugs if the suspect environment settings were present, and the other five would not. One of the eight configuration files (three with problematic settings and five with only good settings) and one input is assigned to each of the 87 user profiles randomly. We assume by default that 20 profiles include environment settings and input that can activate a bug, whereas 67 of them do not. Some of the 67 profiles may have failure-inducing input, but not the environment settings that can activate the bug.

To mimic the situation where some users have failure inducing settings, but do not observe the failure or misbehavior because they do not have the input that triggers the bug, we perform three types of experiments: $perfect$, $imperfect60$, and $imperfect20$. Table 3.3 enumerates the experiments, the type of values that the environment settings are assigned, and the number of failure-inducing profiles for each of the experiments. In the $perfect$ case, the 20 profiles with environment settings that can activate the bug are classified as failed profiles, whereas the other 67 are classified as successful ones. As a result, there is 100% correlation between those resources and the failure. This is the best case for the feature selection for environment-related failures, and possibly the recommendation accuracy of Mojave.

In the two imperfect cases, the environment settings are the same as in the perfect

| Experiment | Type of Values | | Profiles | |
|---|---|---|---|---|
| | System Env | Application Specific Env | Failure Inducing | Actually Failed |
| perfect | Real | Real | 20 | 20 |
| imperfect60 | Real | Real | 20 | 12 |
| imperfect20 | Real | Real | 20 | 4 |

Table 3.3: Mojave experiments.

case. However, not all profiles with environment settings that cause the failure are assigned an input that activates the bug, and therefore, not labeled as failures. In particular, only 60% of these profiles are assigned failure-inducing input (and labeled failures) in the $imperfect60$ case, and 20% in the $imperfect20$ case. These scenarios may result in the feature selection picking more SERs for the environment-related failures than in the perfect case.

In all of the experiments, the feature selection step considers the features ranked within 30% of the highest ranked feature as suspects. This step takes $1-3$ seconds for each experiment across all the five bugs. The static analysis step takes $66-124$ seconds to compute the set of suspect routines for the environment-related bugs.

**Learning and recommendation.** We execute the instrumented version of the software (before the upgrade is applied) with the configuration file and the input assigned to collect the call sequence data. The call sequence data and the success/failure flags are then used to calculate the *SSimilarity* and *FSimilarity* measures for all the users. The computation of the two similarity measures requires a quadratic number of pairwise LCS calculations, each of which is quadratic in the length of the sequences. In our experiments, each pairwise LCS computation takes $1-2$ seconds to execute, and the time to compute each similarity measure is at most 120 seconds.

The environmental resources of a single machine, parsed/chunked and fingerprinted, along with the two similarity measures and success/failure flag constitute a single user profile. The 87 user profiles serve as the input to the classification algorithm.

In all of our experiments, we use two-thirds (57) of the profiles as the training data to learn the prediction model, and the remaining one-third (30) as the test data for

| Bug | SERs | SRs | \|CallSequence\| | | \|SCSR\| | | \|$90^{th}$ percentile LCS\| | | Prediction (Model) |
|---|---|---|---|---|---|---|---|---|---|
| | | | Success | Failure | Success | Failure | SSim | FSim | |
| Port | 3 | 22 | 6K-47K | 29K-73K | 275-605 | 380-632 | 79-100 | 80-98 | 1-7 |
| X11 | 3 | 20-21 | 2.7K-81K | 2.8K-2.9K | 99-390 | 104-107 | 28-100 | 50-100 | 2-7 |
| Proxy | 3 | 15 | 538-58K | 797-37K | 70-1.7K | 98-1.4K | 38-100 | 36-99 | 1-11 |
| SQLite | 2-3 | 12-13 | 10K-22K | 1.4K-15K | 19-92 | 24-168 | 54-100 | 34-88 | 5-12 |
| uServer | NA | NA | 829-8K | 811-1.9K | 829-8K | 811-1.9K | 46-100 | 19-95 | 2-8 |

Table 3.4: Results for the learning phase (Port = Port forwarding; X11 = X11 forwarding; Proxy = ProxyCommand; SSim = SSimilarity; FSim = FSimilarity).

the recommendation phase. In our experiments, the classification step takes $22 - 231$ seconds to learn the prediction model. In particular, it takes $22-59$ seconds for the four environment-related bugs, and $129 - 231$ seconds for the bug that is not environment-related (uServer bug).

### 3.5.2 Results

**OpenSSH: Port forwarding bug.** Recall that this bug was introduced in the ssh code by version 4.7. Mojave collects call sequence data using the instrumented version of the software before the upgrade. As shown in the Table 3.4, the length of the call sequences collected by Mojave ranges from 6K to 47K for the success, and 29K to almost 73K for the failed instances. In addition, Mojave identified 101 environmental resources, including the parameters in the configuration files, the operating system and library dependencies, hardware data, and other relevant files. Many of these resources, such as library files, are split into smaller chunks; for others, such as configuration files, each parameter is considered a separate feature. Overall, there are 325 features, forming the input to the feature selection step to determine if the bug is environment-related.

The feature selection step determines that the bug is indeed environment related, and selects three features across all the experiments: configuration parameters `Tunnel`, `BatchMode`, and `RSAAuthentication`. Features BatchMode and RSAAuthentication have three possible values: yes, no, or missing. In the configurations we collected, it so happened that RSAAuthentication was set to yes, and BatchMode to no in two of the three failed profiles, causing them to be highly correlated with the failure. Recall that we did not assign these values; we retrieved the configurations from the Web and changed only the setting of the Tunnel parameter. These three parameters correspond to 8 suspect variables in ssh. The static analysis results in 22 suspect routines. Mojave uses the suspect routines to filter out the call sequence data to contain only the sequence connecting suspect routines (SCSR).

As shown in Table 3.4, the filtering step reduces the length of the sequences by $22 - 115X$ to $275 - 605$ for the users where the upgrade succeeded, and $380 - 632$ for the users where the upgrade failed. This reduction speeds up the call sequence similarity computation significantly. The success and failure similarity measures range from $79 - 100\%$ and $80 - 98\%$, respectively. Mojave updates the user profiles with these similarity measures, and passes 57 of the updated profiles to the classification algorithm.

The classification algorithm outputs the prediction model comprising only 1 feature for the $perfect$ case, 7 for the $imperfect60$ case, and 5 for the $imperfect20$ case. The only feature in the $perfect$ case is the OpenSSH configuration parameter `Tunnel`, the culprit environment feature. In the $imperfect60$ case, in addition to `Tunnel`, `SSimilarity` (the success similarity measure for the call sequence), `FSimilarity` (the failure similarity measure of the call sequence), the OpenSSH configuration parameter `HostbasedAuthentication`, and system environment resources: `cpuinfo`, filesystems, and `kallsyms`. In the $imperfect20$ case, the prediction model includes the two similarity measures: `SSimilarity` and `FSimilarity`, the `Tunnel` OpenSSH configuration parameter, and the system configuration settings `libcom_err` and `cpuinfo`.

Using the learned prediction model, Mojave computes the recommendations for the remaining 30 profiles. Table 3.5 presents the recommendation results for the three experiments. In the $perfect$ and the $imperfect20$ cases, Mojave correctly predicts whether the upgrade will succeed or fail for all new users resulting in 100% recommendation accuracy. In the $imperfect60$ case, it correctly predicts success or failure for all but one user, an accuracy of 97%. Specifically, it incorrectly predicts that

| Bug | Experiment | Training | | Test | | Mojave Accuracy | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **Success** | **Fail** | **Success** | **Fail** | **TP** | **TN** | **FP** | **FN** |
| Port | perfect | 42 | 15 | 25 | 5 | 25 | 5 | 0 | 0 |
| | imperfect60 | 48 | 9 | 27 | 3 | 27 | 2 | 0 | 1 |
| | imperfect20 | 34 | 3 | 29 | 1 | 29 | 1 | 0 | 0 |
| X11 | perfect | 42 | 15 | 25 | 5 | 25 | 5 | 0 | 0 |
| | imperfect60 | 48 | 9 | 27 | 3 | 27 | 3 | 0 | 0 |
| | imperfect20 | 34 | 3 | 29 | 1 | 29 | 1 | 0 | 0 |
| Proxy | perfect | 42 | 15 | 25 | 5 | 25 | 5 | 0 | 0 |
| | imperfect60 | 48 | 9 | 27 | 3 | 25 | 3 | 2 | 0 |
| | imperfect20 | 34 | 3 | 29 | 1 | 29 | 1 | 0 | 0 |
| SQLite | perfect | 42 | 15 | 25 | 5 | 25 | 5 | 0 | 0 |
| | imperfect60 | 48 | 9 | 27 | 3 | 26 | 3 | 1 | 0 |
| | imperfect20 | 34 | 3 | 29 | 1 | 29 | 0 | 0 | 1 |
| uServer | perfect | 42 | 15 | 25 | 5 | 25 | 4 | 0 | 1 |
| | imperfect60 | 48 | 9 | 27 | 3 | 27 | 3 | 0 | 0 |
| | imperfect20 | 34 | 3 | 29 | 1 | 29 | 0 | 0 | 1 |

Table 3.5: Recommendations for OpenSSH (three), SQLite (one) and uServer (one) bugs (Port = Port forwarding; X11 = X11 forwarding; Proxy = ProxyCommand).

the upgrade will pass for the user, but the upgrade failed at that user site. Mojave mis-predicted the failure likelihood for this user because the user's attribute values (values of the similarity measures and the `cpuinfo`) were similar to the other users where the upgrade passed.

**OpenSSH: X11 forwarding bug.** Recall that the X11 forwarding bug affected the sshd program of OpenSSH version 4.2. Mojave identified 123 environmental resources, resulting in a total of 354 features. The feature selection step for the $perfect$ experiment selects 3 features: configuration parameters `X11Forwarding`, `AuthorizedKeysFile`, and `ChallengeResponseAuthentication`. In the $imperfect60$ and $imperfect20$ cases, Mojave selects three features: configuration parameters `X11Forwarding`, `AuthorizedKeysFile`, and `PidFile`. AuthorizedKeysFile and PidFile were assigned the default value in two out of the three failed real user profiles, whereas ChallengeResponseAuthentication was set to no value in two of them. These four features correspond to seven actual variables in sshd.

The static analysis results in 20 suspect routines in the *perfect*, and 21 suspect routines in the *imperfect* cases. Using the *SuspectRoutines*, the filtering step reduces the length of the sequences by $27 - 200X$ to $104 - 107$ for the success instances, and $99 - 390$ for the failed instances. The success and failure similarity measures range from $28 - 100\%$ and $50 - 100\%$, respectively. Mojave then updates the user profiles with the similarity measures, and passes them to the classification algorithm.

The classification algorithm outputs the prediction model comprising 2 features for the *perfect* case, 7 for the *imperfect60*, and 5 for the *imperfect20* case. In the perfect case, `X11Forwarding` (the configuration parameter that activates the bug) and `SSimilarityLoginit` (success similarity measure of the call sequence that starts with `loginit` routine) are the two features in the prediction model. The *imperfect60* case includes the OpenSSH configuration parameters `X11Forwarding` and `AuthorizedKeysFile`, the three similarity measures: `FSimilarityMain` and `SSimilarityMain` (failure and success similarity measures for the call sequence beginning with the `main` routine) and `SSimilarityLoginit` (success similarity measure for the call sequence starting with the `loginit` routine), and `HOSTNAME` and `swaps` system environment resources. The *imperfect20* case includes `libcom_err` and `cpuinfo` in addition to the three similarity measures: `SSimilarityMain`, `SSimilarityLoginit`, and `FSimilarityLoginit`. Presence of the similarity measures in the model imply that the similarity between the new user's program behavior and the existing users' program behavior is a highly probable failure (or success) predictor.

Using the learned prediction model, Mojave computes the recommendations for the 30 test profiles. Mojave correctly predicts the upgrade's success or failure for all the new users, an accuracy of 100%.

**OpenSSH: ProxyCommand bug.** This bug affected ssh in version 4.9. We performed the same 3 experiments with this upgrade.

The feature selection step produces 3 SERs and static analysis produces 15 suspect routines. Filtering the call sequences with these suspect routines reduces the size of the success sequences from $538 - 58K$ to $70 - 1.7K$, and failed sequences from $797 - 36K$

to $98 - 1.4K$, a reduction of $7 - 34X$. The success and fail similarity measures range from $38 - 99\%$ and $36 - 94\%$, respectively. Mojave then updates the user profiles with the similarity measures, and passes them to the classification algorithm.

The classification algorithm outputs the prediction model comprising 1 feature for the $perfect$ case, 7 for the $imperfect60$ case, and 8 for the $imperfect20$ case. In the $perfect$ case, `SHELL` (the environment resource that activates the bug) is the only feature. The $imperfect60$ case includes `FSimilarity` (failure similarity measure) and `SSimilarity` (success similarity measure), configuration parameters (*Host* and *HostbasedAuthentication*), system resources (*cpuinfo* and `libcom_err`) in addition to `SHELL`. The $imperfect20$ case includes `kallsyms` in addition to the seven features in the $imperfect60$ case. Presence of the `SHELL` and the two similarity measures in the model imply that the suspect environment and the similarity between the new user and the current users are failure (or success) predictors.

In the $perfect$ case, Mojave correctly predicts whether the upgrade will succeed or fail for all new users resulting in 100% recommendation accuracy. In the $imperfect60$ case, it correctly predicts success or failure for all but two users, an accuracy of 93%. Specifically, it incorrectly predicts that the upgrade will fail for those users, however, the upgrade did not fail at those user sites. In the $imperfect20$ case, it provides correct recommendation for the 29 users where the upgrade would succeed, but mis-recommends for the one user where the upgrade would have failed. Mojave missed identifying the potential upgrade failure for because there are very few failing instances to accurately learn the failure predictors.

**SQLite bug.** We injected this bug in SQLite version 3.6.14.2, which comprises 67K LOC. The results for the three experiments show that feature selection identified 2-3 SERs, and the static analysis produced 12-13 *SuspectRoutines*. Using the *SuspectRoutines*, the filter shortens the length of call sequences from $10 - 22K$ to $19 - 92$ for the users where the upgrade passed, and from $1.4 - 14K$ to $24 - 168$ for the users where the upgrade failed, a reduction of $57 - 500X$ over the original call sequences. The similarity measures for the updated call sequences is $54 - 100\%$ and

$34 - 88\%$ respectively. The environment data along with the similarity measures and the success/fail flags are passed to the classification algorithm.

The classification algorithm outputs the prediction model comprising 4 features for the $perfect$ case, 9 for the $imperfect60$ case and 4 for the $imperfect20$ case. In the $perfect$ case, the features chosen are the three configuration parameters *Echo*, *Bail* and *Batch*; and *FSimilarity* similarity measure. The $imperfect60$ case includes the system resources *cpuinfo*, *libc.so*, *kallsyms*, *scsi* and *swap* in addition to the five features in the perfect case. The $imperfect20$ case includes the configuration parameter *Bail*, *FSimilarity* similarity measure, and system resources *cpuinfo* and *libc.so*.

Using the learned prediction model, Mojave computes the recommendations for the remaining 30 profiles. In the $perfect$ case, Mojave correctly predicts whether the upgrade will succeed or fail at the new user site for all the users, an accuracy of 100%. In the $imperfect60$ case, it misclassified one success instance as failed, and in the $imperfect20$ case misclassified one failed instance as success, an accuracy of 97% for both cases.

**uServer bug.** We injected this bug in uServer version 0.6.0, which comprises 37K LOC. Again, we ran the three experiments: $perfect$, $imperfect60$ and $imperfect20$. Since this bug is not environment related, ranks of the top-ranked features consistently exhibit high standard deviations at the feature selection step. Thus, feature selection properly flags this bug as unrelated to the environment, and therefore, Mojave skips the static analysis and filter (environment-related optimization) steps, and moves on to the call sequence similarity step. The output of the call sequence similarity step are the two similarity measures: *SSimilarity* ranges from $46 - 100\%$ and *FSimilarity* from $19-95\%$. The similarity measures along with the environment data and the success/fail flags serve as the input to the classification step.

The classification step outputs the prediction model comprising 6 for the $perfect$ case, 6 for the $imperfect60$ case and 2 for the $imperfect20$ case. The perfect case includes the two similarity measures *FSimilarity* and *SSimilarity* and system resources *cpuinfo*, *iomem*, `kallsyms`, *libcom_err.so* and *mounts*. The $imperfect60$ case includes the two similarity measures, *cpuinfo* , *kallsyms*, *libcom_err.so*, *devices* and *mounts*. The $imperfect20$ case comprises *liberr_com.so* and *cpuinfo*.

In the $perfect$ case, Mojave correctly predicts whether the upgrade will succeed or fail for 28 out of 30 new users resulting in 93% recommendation accuracy. Specifically, it mis-predicts one success and failed instance. In the $imperfect60$ case, it correctly predicts success or failure for all the users, an accuracy of 100%. In the $imperfect20$ case, it provides correct recommendation for the 29 users where the upgrade would succeed, but mis-recommends for the one user where the upgrade would have failed. Mojave missed identifying the potential upgrade failure for because there are very few failing instances to accurately learn the failure predictors.

Again, although trivial, the SQLite and uServer results illustrate that Mojave can be used without modification to provide accurate recommendations for a variety of applications.

**Summary.**The results for the five bugs indicate that Mojave provides recommendations with very high accuracy ($93 - 100\%$) for all types the upgrade failures (both environment-related and not environment-related), and across different applications. Therefore, we expect that Mojave would prevent most upgrade failures in the field.

# Chapter 4
# Sahara: An Upgrade Debugging System

## 4.1 Introduction

Sahara simplifies the debugging of environment-related upgrade problems by pinpointing the subset of routines and variables that are most likely the source of misbehavior [4]. Sahara's design was motivated by two observations: (1) since the problem was caused by one or more aspects of the user environment, it is critical to identify these suspect aspects and their effects throughout the code; and (2) since the previous version of the software behaved properly, it is critical to identify the behavioral differences between the previous and upgraded versions.

Given these observations, the root cause of an upgrade problem is most likely to be in the code that is both (1) affected by the suspect aspects of the environment and (2) whose behavior has deviated after the upgrade. To isolate this code, Sahara combines information collected from many users of the software, machine learning techniques, static and dynamic source analyses. The machine learning and the static analysis run at the developer's site, whereas the data collection and dynamic analysis run at the users' sites (for those users who are willing to run Sahara). Sahara can be used in isolation or with Mojave (to debug failures observed by the existing users, and for the cases where new users decide to upgrade, despite the fact that Mojave recommended against the upgrade).

In more detail, Sahara applies feature selection [53] on the environment and upgrade success/failure information received from users to rank the aspects of the environment that are most likely to be the source of the misbehavior. Then, it uses def-use static analysis [1] to identify the set of variables whose values derive directly or indirectly from the suspect aspects. The routines in which these variables are used become the first set of potential culprits. Note that the feature selection and the static analysis steps are the same as described in Section 3.4. At this point, Sahara deploys instrumented versions of the current and upgraded codes to the user sites that reported misbehaviors. It then runs the instrumented versions automatically (and with the same inputs) to collect information about all routine calls and returns. Using this information, it uses value spectra [54] to identify the set of routines that caused the behavior to deviate from one execution to the other at each misbehaving site. These sets of routines are also considered suspects. Finally, Sahara intersects the sets of suspect routines resulting from the static and dynamic analyses; those in the intersection should be debugged first.

To evaluate Sahara, we study three real upgrade problems with the OpenSSH suite, one synthetic problem in the SQLite database engine, and one synthetic problem with the uServer Web server described in Section 3.5.1. For each problem and user information scenario, we isolate the number of routines that changed in the upgrade, and the number of routines identified through static analysis alone, dynamic analysis alone, and full-blown Sahara. Our results demonstrate that Sahara produces an offender list that always includes the routines responsible for the bugs. The exact number of routines in the offender list depends on the characteristics of the information received from users. In experiments where we varied these characteristics widely, Sahara recommends 2-21 suspect routines that should be debugged first. These numbers can be 20x smaller than the number of routines affected by the upgrades. Compared to static and dynamic analyses alone, Sahara reduces the numbers of suspect routines by 1.4x-6x and 14x-40x, respectively. Given its accuracy and these large reductions, we expect that Sahara can significantly reduce debugging time in practice.

## 4.2 A Motivating Example

To make our exposition more concrete, let us look at a simple example in Listing 4.1. The example takes the name of an environment variable as input using a call to *getenv*() (line 18). It then checks if the length of the string is smaller than or equal to 9 (line 4). Depending on the outcome of the comparison, a different output is produced (lines

```
 1 int env2 = 0, glob = 3;
 2
 3 int checklength(int len) {
 4   if (len <= 9) % Upgrade changes sign to <
 5     return len;
 6   else
 7     return −1;
 8 }
 9 int secondfunction(float a) {
10   int ai = ceil(a);
11   if ((glob + ai) < 5)
12     return 100;
13   else
14     return 10;
15 }
16 int main() {
17   char uname[80];
18   strcpy(uname, getenv("SHELL"));
19   env2 = strlen(uname);
20   int retval1 = checklength(env2);
21   if (retval1 > 0)
22     printf("Out1:%d",secondfunction(2.2));
23   else
24     printf("Out2:%d",secondfunction(5.1));
25   return 0;
26 }
```

Listing 4.1: Sahara: Example program.

21-24).

Let us assume that the upgrade simply changes the sign in line 4 from “<=” to “<”. This upgrade will fail at user sites where the `$SHELL` variable is set to `/bin/bash` or `/bin/tcsh`, but not `/bin/csh` or `/bin/ksh`, for instance. More generally, the upgrade will fail where the length of the value of the `$SHELL` environment variable is exactly 9. However, the program ran successfully at these sites before the upgrade. This upgrade failure is similar to the ProxyCommand bug [43], and a variation of this bug was detailed in Section 3.5.1. Note that the two examples: Listing 4.1 and Listing 3.1 have minor differences. We created two different variations to enable simpler explanation of their respective systems. Specifically, the example in Listing 4.1 does not take any input, and is dependent only on the user's environment.

The failure for the upgraded code in Listing 4.1 has two interesting characteristics. First, the upgrade fails only at a subset of user sites, which may have been the reason the bug went undetected during development. Second, despite the fact that the two versions of the code are input-compatible, the execution behavior changes with the upgrade both in terms of the path executed and the output produced.

Given these characteristics, identifying the aspects of the environment that correlate with the failure is a necessary first step for efficiently diagnosing the failure. In this simple example, the name of the shell is the aspect of the environment that triggers the failure. It is also important to identify the variables and routines in the code that are directly or indirectly affected by the environment. Note that the name of the shell is initially assigned to the `uname` array; only later does variable `env2` become related to the environment. Thus, variables `uname` and `env2`, as well as routines `main` and `checklength` are suspect. However, identifying these suspects is not sufficient, because the program behaved correctly before the upgrade was applied in the same environment. We also need to determine how the upgraded version of the program has deviated from the current version. This analysis would then show that routine `checklength` and `secondfunction` behave differently in the two versions, meaning that they are also suspects. The root cause of the failure is most likely to be contained in the code that is affected by *both* the suspect environment and whose behavior has changed after the upgrade, i.e. routine `checklength`. This routine is exactly where the bug is in our example.

## 4.3 Design and Implementation

**Overview.** Figure 4.1 illustrates the steps involved in Sahara. The upgrade deployment, execution at the user's site with his/her input, gathering the user's environment data and success/failure flags, and transferring the user data back to the developer site (steps 1-3) in Sahara are similar to the corresponding steps in Mojave. Note that a failure flag may mean that the upgrade did not install or execute properly, crashed or misbehaved itself, or resulted in another software to misbehave [12].

In case the upgrade misbehaved at one user site at least, Sahara gathers the user's environment settings and success/failure information from the users, and performs machine learning on the this data to determine the aspects of the environment that

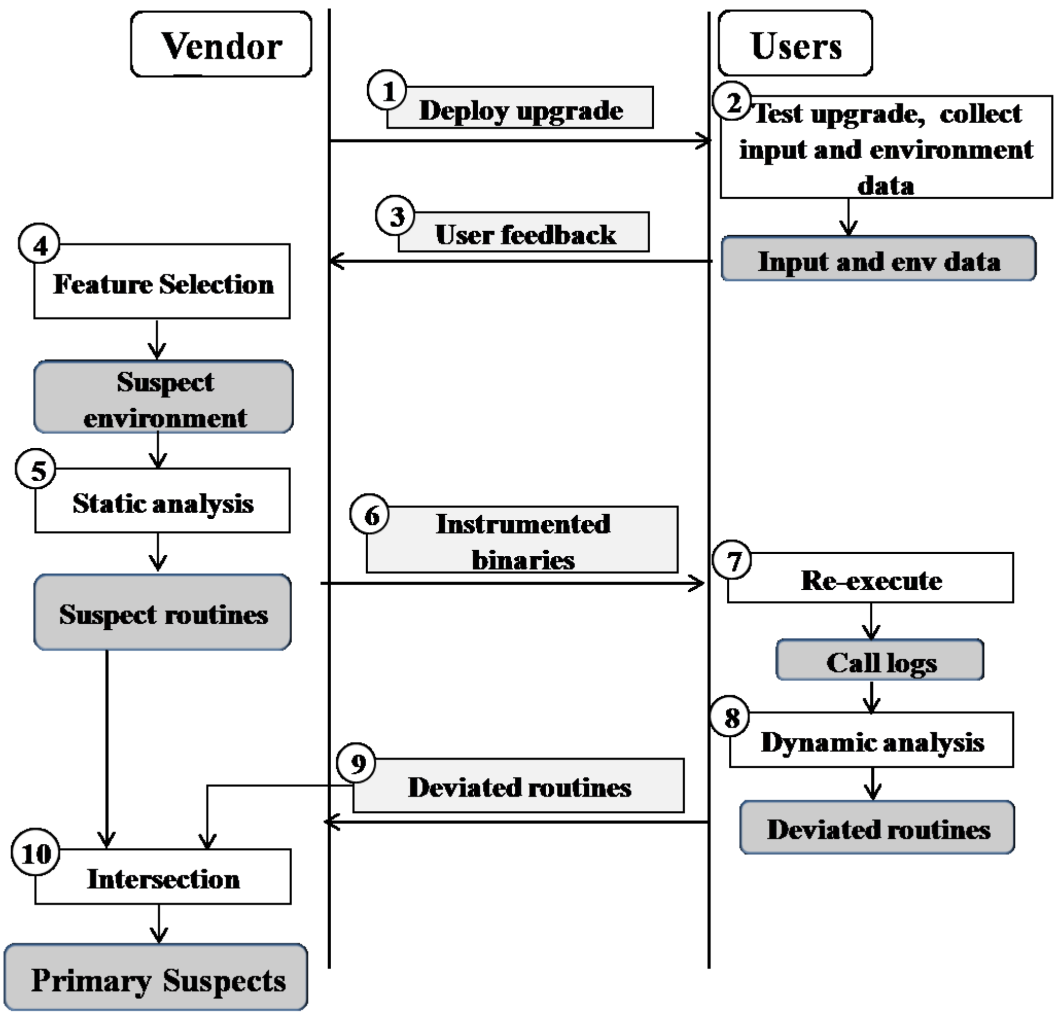


Figure 4.1: Overview of Sahara.

are most likely to have caused the misbehavior (step 4). Thereafter, Sahara isolates the variables in the code that derive directly or indirectly from those suspect aspects using def-use static analysis. The routines that use these variables are considered suspect (step 5). Steps 4-5 of Sahara are similar to steps 4-5 of Mojave (Section 3.4).

Sahara then deploys instrumented versions of the current and upgraded codes to the user sites that reported failures (step 6). At each of those sites, Sahara executes both versions with the inputs collected in step 2 and collects dynamic routine call/return information (step 7). Sahara then compares the logs from the two executions to determine the routines that exhibited different dynamic behavior (step 8). This step is done at the failed user sites to avoid transferring the potentially large execution logs back to the developer's site. Sahara then transfers the list of routines that deviated at each failed user site back to the developer's site (step 9); the routines on these lists are considered suspect as well.

Finally, Sahara intersects the suspects from the static and dynamic analyses (step 10). It reports the intersection to the developer as the routines to debug first. If the problem is not found in this set, other suspect routines should be considered.

Next, we detail the implementation of these steps.

**Upgrade deployment, tracing, and user feedback (steps 1-3).** The upgrade deployment, tracing, and the user feedback are similar to steps $1 - 3$ in Mojave as described in Section 3.4. The only difference is that Sahara does not rely on the execution behavior (call sequence) data from the users; Sahara transfers only the user's environment settings and the success/failure data back to the developer site.

This data represents the profile of the corresponding user site. After the first several executions, Sahara turns its data collection off to minimize overheads. User profiles from all sites serve as the input to the feature selection step. Section 4.5 systematically studies the impact of user profiles with various characteristics.

**Feature selection (step 4).** Based on the information received from the user sites, this step selects environment resources (called features) with the strongest correlation to the observed upgrade failures. This step is the same as step 4 of Mojave, and is described in Section 3.4.

For the example of Listing 4.1, the root feature chosen by the feature selection would be the SHELL environment variable. The subsets that include SHELL strings of length different than 9 are successes, whereas those that have strings of exactly 9 characters are failures.

Note that as in step 4 of Mojave, Sahara considers all the features that have Gain Ratios within 30% of the highest ranked feature as *Suspect Environment Resources (SERs)*. These SERs serve as input to the static analysis step. We assess the impact of the accuracy of the feature selection step in Section 4.5.

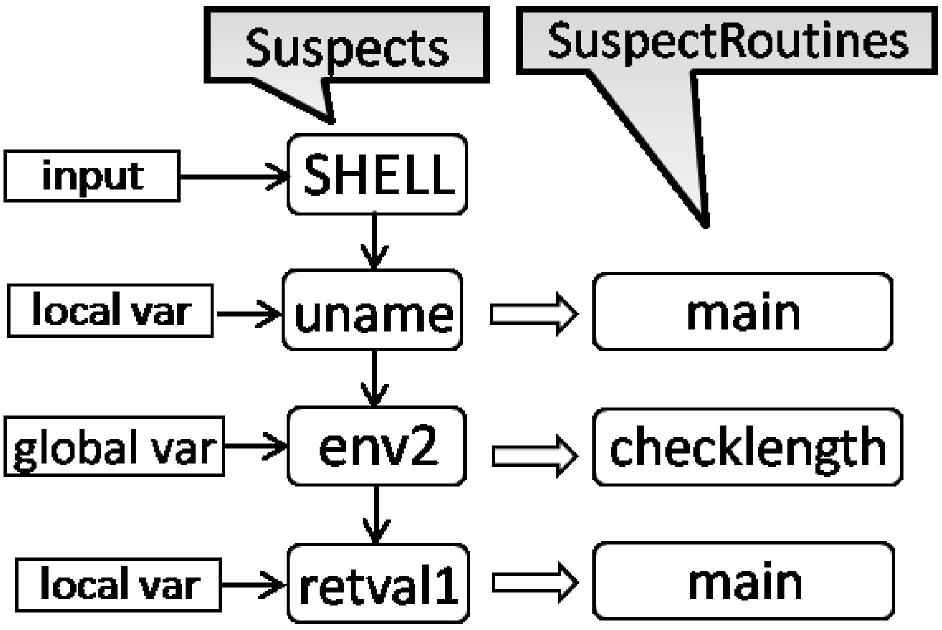


Figure 4.2: Def-use chain, suspect variables and routines for Sahara's simple example.

**Static analysis and suspect routines (step 5).** The static analysis step in Sahara is the same as step 5 of Mojave (Section 3.4). Sahara uses the same two CIL [38] modules: *call-graph* and *def-use* to create def-use chains [1] for each SER. A def-use chain links all the variables that derive directly or indirectly from one SER. Each array is handled as a single variable, whereas struct and union fields are handled separately. Figure 4.2 shows the def-use chain (thin arrows) for our example program.

Similar to step 5 of Mojave, this step outputs *SuspectRoutines (SRs)*, a set of routines that are highly correlated with the failures. For the example in Listing 4.1, `main` and `checklength` are the two suspect routines. The block arrows in Figure 4.2 show why these routines were included as suspects.

**Creating and distributing instrumented versions (step 6).** After the *SRs* are identified, Sahara generates the instrumented versions of the current and upgraded versions of the software.

Sahara uses CIL to automatically instrument the application. The instrumentation is introduced by two new CIL modules, *instrument-calls* and *ptr-analysis.* The instrument-calls module inserts calls to our C runtime library to log routine signatures for all the routines executed in a particular run. A routine's signature consists of the number, name, and values of its parameters, its return value, and any global state that is accessed by the routine. The global state comprises the number, name, and values of all the global variables accessed by the routine. This module works well for logging parameters of basic data types. However, in order to correctly log pointer variables and variables of complex data types, we have implemented the ptr-analysis module. This module inserts additional calls to our C library to track all heap allocations and deallocations.

**Re-execution, value spectra analysis, and deviated routines (steps 7-9).** As we do not want to transfer inputs or large logs across the network, these steps are performed at the failed users' sites themselves. To do so, Sahara first deploys infrastructure to those sites that is responsible for re-execution and value spectra analysis. It then transfers the instrumented binaries of the current and upgraded versions.

Sahara leverages Mirage's re-execution infrastructure, which has been detailed in [12]. This infrastructure executes the instrumented binaries of both versions at the failed user sites, feeding them the same inputs that had caused the upgrade to fail. These inputs were collected in the logs recorded during step 2. To allow for some level of non-determinism during re-execution, Sahara maps the recorded inputs to the appropriate input operations (identified by their system calls and thread ids), even if they are executed in a different order in the log.

As the instrumented versions execute, their dynamic routine call/return information is collected. Listing 4.2 shows the log for the current version, whereas Listing 4.3 does so for the upgraded version of the program.

With these routine call/return logs, Sahara determines the set of routines, called *DeviatedRoutines*, whose dynamic behavior has deviated after the upgrade. Specifically, we implement *fDiff*, a diff-like tool that takes two execution logs as input, and

```
 1 Function main numArgs 0
 2 Globals at ENTRY: 0
 3
 4   Function checklength numArgs 0
 5   Globals at ENTRY: 1
 6   Global: env2 Size: 4 Type: int Value: 9
 7
 8   Globals at EXIT: 1
 9   Global: env2 Size: 4 Type: int Value: 9
10
11   Return: retVal Size: 4 Type: int Value: 9
12
13   Function secondfunction numArgs 1
14   Globals at ENTRY: 1
15   Global: glob Size: 4 Type: int Value: 3
16
17   Parameter: a Size: 4 Type: float Value: 2.2
18
19   Globals at EXIT: 1
20   Global: glob Size: 4 Type: int Value: 3
21
22   Return: retVal Size: 4 Type: int Value: 10
23
24 Globals at EXIT: 0
25
26 Return: retVal Size: 4 Type: int Value: 0
```

Listing 4.2: Execution log of current version.

converts each of them into a sequence of routine signatures. It uses the longest common subsequence algorithm to compute the difference between the two sequences of signatures. A routine has deviated, if one or more of the following differs between the two versions: (1) the number of arguments passed to it; (2) the value of any of its arguments; (3) its return value; (4) the number of global variables accessed by it; or (5) the value of one or more global variables accessed by it. This notion of deviation is similar to that proposed for value spectra [54].

In Listings 4.2 and 4.3, two routines have deviated: `checklength` and `second-function`. The return value of `checklength` changed from 9 before the upgrade to -1 after the upgrade (line 17 in the two figures). The argument to `secondfunction` changed from 2.2 to 5.1 (line 17).

Sahara transfers the *DeviatedRoutines* list to the developer's site for the final step.

**Intersection and list of primary suspects (step 10).** Finally, Sahara computes the union of the *DeviatedRoutines* from the failed user sites. It then intersects this

```
Function main numArgs 0
Globals at ENTRY: 0

  Function checklength numArgs 0
  Globals at ENTRY: 1
  Global: env2 Size: 4 Type: int Value: 9

  Globals at EXIT: 1
  Global: env2 Size: 4 Type: int Value: 9

  Return: retVal Size: 4 Type: int Value: −1

  Function secondfunction numArgs 1
  Globals at ENTRY: 1
  Global: glob Size: 4 Type: int Value: 3

  Parameter: a Size: 4 Type: float Value: 5.1

  Globals at EXIT: 1
  Global: glob Size: 4 Type: int Value: 3

  Return: retVal Size: 4 Type: int Value: 10

Globals at EXIT: 0

Return: retVal Size: 4 Type: int Value: 0
```

Listing 4.3: Execution log of upgraded version.

larger set with the *SRs*. The intersection forms the set of *Prime Suspect Routines (PSRs)*, i.e. the routines most likely to contain the root cause of the failure. For the example, `checklength` is the prime suspect, despite the fact that all 3 routines have some relationship to the users' environment. The root cause is indeed `checklength`.

## 4.4 Discussion

**Sahara and other systems.** Sahara simplifies the debugging of upgrades that fail due to the user environment. As such, Sahara is less comprehensive than systems that seek to identify more classes of software bugs (e.g., [51]). However, Sahara takes advantage of its narrower scope to guide failed upgrade debugging more directly towards environment-related bugs (which are the most common in practice [12]).

In essence, *we see Sahara as complementary to other systems.* In fact, an example combination of systems is the following. Steps 1-4 of Sahara would be executed first. If the user environment is likely the culprit (as determined by the output of step 4), the other steps are executed. Otherwise, another system is activated.

**Dealing with multiple bugs.** The feature selection algorithm is the only part of Sahara that could be negatively affected by an upgrade with multiple bugs. The other components of Sahara are unaffected because (1) information about each execution (the resource fingerprints and a success/failure flag) represents at most one bug, (2) static analysis is independent of the number of bugs, (3) each dynamic analysis finds deviations associated with a single bug, and (4) the union+intersection step is independent of the number of bugs.

Sahara is effective when faced with multiple bugs, even when feature selection does not produce the ideal results. To understand this, consider the two possible scenarios: (1) all bugs are environment-related; and (2) one or more bugs are unrelated to the environment.

When all bugs are environment-related and involve the same environment resources, feature selection works correctly and Sahara easily produces the prime suspects for all bugs. If different bugs relate to different sets of environment resources, feature selection could misbehave. In particular, if there is not enough information about all bugs, feature selection could mis-rank the environment resources that are relevant to the less frequent bugs to the point that they do not become SERs. This would cause the remaining steps to eventually produce the prime suspects for the more frequent bugs only. After those bugs are removed, Sahara can be run again to tackle the less frequent bugs. This second time, feature selection would rank the environment resources of the remaining bugs more highly. Other systems rely on similar multi-round approaches for dealing with multiple bugs, e.g. [19].

When one or more bugs are not related to the environment, feature selection could again misbehave if there is not enough information about the bugs that are environment-related. This scenario would most likely cause feature selection to low-rank all environment resources. In this case, the best approach is to resort to a different system, as discussed above. In contrast, if there is enough information about the environment-related bugs, feature selection would select the proper SERs.

Despite this good behavior, the dynamic analysis at some failed sites would identify *DeviatedRoutines* corresponding to bugs that are not related to the environment. However, those routines would not intersect with those from the static analysis, leading to the proper prime suspect results.

**Limitations of Sahara's current implementation.** *Sahara currently implements simple versions of its components.* As a proof-of-concept, the goal of this initial implementation is simply to demonstrate how to combine different techniques in a useful and novel way. However, as we discuss below, more sophisticated components can easily replace the existing ones.

Sahara shares the limitations of the upgrade deployment, tracing infrastructure, collection and transfer of the user feedback, and static analysis with Mojave, as described in Section 3.4.3.

Sahara employs dynamic analysis (along with static analysis) to narrow the set of routines that are likely to contain the root cause of the failure. However, most (or even all) routines could be found to deviate from their original behaviors. Fortunately, these worst-case scenarios are extremely unlikely in a single upgrade.

Execution replay at the failed sites is currently performed without virtualization. Using virtual machines would enable us to automatically handle applications that have side-effects, but at the cost of becoming more intrusive and transferring more data to the failed sites. Sahara can be extended to use replay virtualization. On the positive side, Sahara performs a single replay at a failed site, which is significantly more efficient than the many replays of techniques such as delta debugging [58].

Our current approach for handling replay non-determinism is very simple: Sahara tries to match the recorded inputs to their original system calls when re-executing each version of the application. Internal non-determinism (e.g., due to random numbers or race conditions) is currently not handled and may mislead the dynamic analysis if it changes: the number or value of the arguments passed to any routines, the number or value of the global variables they touch, or their return values. Sahara can be combined with existing deterministic replay systems to eliminate these problems.

Finally, Sahara guides the debugging process by pinpointing a set of routines to debug first. Pinpointing a single routine or a single line causing the failure may not even be possible, since the root cause of the failure may span multiple lines and routines. Moreover, the systems that attempt such pinpointing (e.g., [27, 51, 58]) often incur substantial overhead at the users' sites, such as running instrumented code all the time, check-pointing state at regular intervals, and multiple replays.

## 4.5 Evaluation

In this section, we describe our methodology and evaluate Sahara with the same five bugs as Mojave: three real bugs in OpenSSH (Section 2.6), a synthetic bug in SQLite, and a synthetic bug in uServer (Section 3.5.1).

### 4.5.1 Methodology

**Upgrade deployment.** Sahara uses the same tracing infrastructure, parsers (Table 3.2) and the upgrade deployment setup as Mojave. Recall that we collected environment data from 87 machines across two clusters. Table 3.1 denotes the experimental setup parameters and their default values. Note that a single user profile comprises the environmental resources of a single machine (parsed/chunked and fingerprinted), and the success/failure flag.

**User site environments.** To evaluate Sahara's behavior in the various real-world like scenarios, we perform six types of experiments: *random_perfect*, *random_imperfect60*, *random_imperfect20*, *realconfig_perfect*, *realconfig_imperfect60*, and *realconfig_imperfect20*. As shown in the Table 4.1, for the random_perfect experiments, the values of all the environment resources related to the application are chosen at random, except for the resources that relate directly to the bug. Moreover, the 20 profiles with environment settings that can activate the bug are classified as failed profiles, whereas the other 67 are classified as successful ones. As a result, there is 100% correlation between those resources and the failure. This is the best case for feature selection in Sahara, as it finds the minimum set of SERs.

| Experiment | Type of Values | | Profiles | |
|---|---|---|---|---|
| | System Env | Application Specific Env | Failure Inducing | Actually Failed |
| random_perfect | Real | Random | 20 | 20 |
| random_imperfect60 | Real | Random | 20 | 12 |
| random_imperfect20 | Real | Random | 20 | 4 |
| realconfig_perfect | Real | Real | 20 | 20 |
| realconfig_imperfect60 | Real | Real | 20 | 12 |
| realconfig_imperfect20 | Real | Real | 20 | 4 |

Table 4.1: Sahara experiments.

In the two random_imperfect cases, the environment settings are the same as in the random_perfect case. However, not all profiles with environment settings that cause the failure are labeled as failures. In particular, only 60% of these profiles are labeled failures in the random_imperfect60 case, and only 20% in the random_imperfect20 case. These imperfect experiments mimic the situation where some users simply have not activated the bug yet, possibly because they have not exercised the part of the code that uses the problematic settings. These scenarios may lead feature selection to pick more SERs than in the random_perfect case.

In the three types of experiments described above, the application-related environment includes random values. The three realconfig scenarios are similar to the experiments specified in the Section 3.5.1, and listed in the Table 3.3 under the names $perfect$, $imperfect60$, and $imperfect20$. Recall that in the realconfig_perfect case, all the 20 profiles with problematic settings are labeled as failures, whereas the 67 others are labeled as successful. In the realconfig_imperfect60 and realconfig_imperfect20 experiments, only 60% and 20% of the profiles with these settings are labeled as failures, respectively. The realconfig experiments are likely to lead to more SERs than the random ones. We do not study realconfig scenarios for SQLite because the bug we inject into it is synthetic.

In the six types of experiments described above, we assume that there are 20 users with problematic settings for the OpenSSH-related environment. To assess the impact of having different numbers of sites with these bad settings, we consider four more types of experiments: *random_perfect30*, *random_perfect10*, *realconfig_perfect30*, and *realconfig_perfect10*. The 30 and 10 suffixes refer to the number of profiles that exhibit the environment settings that can cause the upgrades to fail.

As stated in Table 3.1, we consider the features ranked within 30% of the highest ranked feature as suspects in all of our experiments. In addition, we use inputs that we know will activate the bugs.

**Overheads.** The overhead of tracing and feature selection in Sahara is similar to that in Mojave (Section 3.5.1). The static analysis step in Sahara takes $21 - 27$ seconds. This overhead differs from that in Mojave because Sahara analyzes the post-upgrade versions of the software, whereas Mojave analyzes the pre-upgrade versions. The time taken by the dynamic analysis step in Sahara ranges from $5 - 109$ seconds for the five bugs.

### 4.5.2 Results

**OpenSSH: Port forwarding bug.** Recall that this bug was introduced in the ssh code by version 4.7. This version has 58K LOC and 1529 routines (729 routines in ssh). The `diff` between versions 4.6 and 4.7 comprises approximately 400 LOC and 65 routines. Sahara identified 101 environmental resources, including the parameters in the configuration files, the operating system and library dependencies, hardware data, and other relevant files. Many of these resources, such as library files, are split into smaller chunks; for others, such as configuration files, each parameter is considered a separate feature. Overall, there are 325 features, forming the input to the feature selection step.

Table 4.2 shows the results for each of the analyses in Sahara and all techniques combined for every experiment. The feature selection step results in merely 1 feature (out of 325) chosen as suspect in the random_perfect, random_imperfect60, and random_imperfect20 cases. In these experiments, the environment resource that is actually determinant in the failures, configuration parameter `Tunnel`, was the only suspect because the other environmental resources were assigned random values in all user profiles. This resulted in a very high correlation between the failure and this resource, even in the random_imperfect cases. The Tunnel parameter corresponds to 4 suspect variables in ssh.

| **Bug** | **Experiment** | **dRs** | **SERs** (feature selection) | **SRs** (static analysis) | **DRs** (dynamic analysis) | **Primary suspects (Sahara)** |
|---|---|---|---|---|---|---|
| Port | random_perfect | 65 | 1 | 12 | 124 | 6 |
| | random_imperfect60 | 65 | 1 | 12 | 124 | 6 |
| | random_imperfect20 | 65 | 1 | 12 | 124 | 6 |
| | realconfig_perfect | 65 | 3 | 22 | 124 | 7 |
| | realconfig_imperfect60 | 65 | 3 | 22 | 124 | 7 |
| | realconfig_imperfect20 | 65 | 3 | 22 | 124 | 7 |
| X11 | random_perfect | 137 | 1 | 18 | 157 | 6 |
| | random_imperfect60 | 137 | 1 | 18 | 157 | 6 |
| | random_imperfect20 | 137 | 1 | 18 | 157 | 6 |
| | realconfig_perfect | 137 | 3 | 21 | 157 | 7 |
| | realconfig_imperfect60 | 137 | 3 | 20 | 157 | 6 |
| | realconfig_imperfect20 | 137 | 3 | 20 | 157 | 6 |
| Proxy | random_perfect | 122 | 2 | 10 | 64 | 7 |
| | random_imperfect60 | 122 | 2 | 10 | 64 | 7 |
| | random_imperfect20 | 122 | 2 | 10 | 64 | 7 |
| | realconfig_perfect | 122 | 3 | 15 | 64 | 11 |
| | realconfig_imperfect60 | 122 | 3 | 15 | 64 | 11 |
| | realconfig_imperfect20 | 122 | 3 | 15 | 64 | 11 |

Table 4.2: Results for three OpenSSH bugs (Port = Port forwarding; X11 = X11 forwarding; Proxy = ProxyCommand; dRs = `diff` Routines; SRs = Suspect Routines; DRs = Deviated Routines).

In contrast, in the realconfig_perfect, realconfig_imperfect60 and realconfig_imperfect20 experiments, 3 features are selected: configuration parameters `Tunnel`, `BatchMode`, and `RSAAuthentication`. Features BatchMode and RSAAuthentication have 3 possible values: yes, no, or missing. In the real configurations we collected, it so happened that RSAAuthentication was set to yes, and BatchMode to no in two of the three failed profiles, causing them to be highly correlated with the failure. Recall that we did not assign these values; we retrieved the configurations from the Web and changed only the setting of the Tunnel parameter. These three parameters correspond to 8 suspect variables in ssh.

The static analysis results in 12 suspect routines in the random cases, and 22 in the realconfig cases. The 12 routines comprise those that (1) read the configuration file (*main* and *process_config_line*) and initialize the environment of the ssh client (*initialize_options* and *fill_default_options*); (2) create, enable, or disable a tunnel (*tun_open* and *a2tun*); (3) place the tunnel data into a buffer or a packet (*buffer_put_int* and *packet_put_int*); and (4) enable the port forwarding over this tunnel and create a channel for it (*ssh_init_forwarding*, *channel_new*, *client_request_tun_fwd*, and *clear_forwardings*). Routine channel_new contains the root cause of this failure.

In the realconfig cases, the same 12 routines are suspect, in addition to those affected by RSAAuthentication (*check_host_key*, *confirm*, *key_free*, *key_sign*, *load_identity_file*, *ssh_userauth1*, *try_challenge_response_authentication*, *try_-password_authentication*, *try_rsa_authentication*, and *userauth_pubkey*). BatchMode is used only during the initialization in ssh, so it does not produce other suspects.

The dynamic analysis identifies 124 routines whose behavior has deviated when going from version 4.6 to 4.7. Note that the number of deviations is higher than the number of routines that actually changed. The reason is that the command succeeds before the upgrade and many more routines are invoked, as compared to after the upgrade when the command fails. In our fDiff implementation, the routines that were not called after the upgrade are considered deviations.

The intersection of *SuspectRoutines* and *DeviatedRoutines* is only 6 routines in the random cases and 7 routines in the realconfig cases. In the random cases, the four routines pertaining to reading the configuration file and setting up the environment, and two routines pertaining to enabling or disabling the tunnel, were pruned out after intersection; their behavior did not change after the upgrade. In the realconfig perfect case, confirm was the additional routine identified as primary suspect. The 6 or 7 primary suspects reported by Sahara include the actual culprit (routine channel_new).

From the top six rows in Table 4.2, we can see that the number of primary suspects output by Sahara is 2x-3x lower than that by static analysis, 17x-20x lower than that by dynamic analysis, and 9x-10x lower than the number of routines that were modified in the upgrade. Furthermore, we can see that Sahara is resilient to users that do not report their upgrades to have failed despite having problematic settings for the environment resources that cause the failure.

**OpenSSH: X11 forwarding bug.** Recall that the X11 forwarding bug affected the sshd program of OpenSSH version 4.2. This version has 52K LOC and 1439 routines

(856 routines in sshd). The `diff` between versions 4.1 and 4.2 is approximately 900 LOC and 137 routines. Sahara identified 123 environmental resources, resulting in a total of 354 features.

Table 4.2 presents the results for each of the analyses in Sahara and all techniques combined for every experiment. The feature selection step again results in 1 feature (out of 354) chosen as suspect in the random_perfect, random_imperfect60, and random_imperfect20 cases. This feature is exactly the environment resource that is directly related to the bug: configuration parameter `X11Forwarding`. Like before, feature selection for this bug is extremely accurate in the random experiments, due to the way we assigned values to the other environment resources. This feature corresponds to 3 variables in the sshd code.
In the realconfig_perfect experiment, Sahara selects 3 features: configuration parameters `X11Forwarding`, `AuthorizedKeysFile`, and `ChallengeResponseAuthentication`. In the realconfig_imperfect60 and realconfig_imperfect20 cases, Sahara also selects three features: configuration parameters `X11Forwarding`, `AuthorizedKeysFile`, and `PidFile`. AuthorizedKeysFile and PidFile were assigned the default value in two out of the three failed real user profiles, whereas ChallengeResponseAuthentication was set to no value in two of them. These four features correspond to seven actual variables in sshd.

The static analysis results in 18 suspect routines in the random_perfect and random_imperfect cases, 21 in realconfig_perfect, and 20 in the realconfig_imperfect cases. The 18 routines comprise those that: (1) read the configuration file (*auth_clear_options* and *auth_parse_options*) and initialize the environment of sshd (*initialize_server_options* and *fill_default_server_options*); (2) authenticate the incoming client connection with the options specified and setup the connection (*do_authenticated1*, *do_child*, *do_exec*, *do_exec_pty*, *do_exec_no_pty*, and *do_login*); (3) start a packet for X11 forwarding (*packet_start*); and (4) setup X11 forwarding, create the channel, process X11 requests, and do the cleanup (*server_input_channel_req*, *session_input_channel_req*, *server_input_channel_req*, *session_x11_req*, *session_setup_x11fwd*, *session_close*, and *disable_forwarding*).

In the realconfig cases, all the 18 routines mentioned above are suspect, in addition to those affected by AuthorizedKeysFile (*authorized_keys_file* and *expand_authorized_keys*) and ChallengeResponseAuthentication (*do_authentication2*). PidFile did not result in additional suspect routines, because it is used once in the initialization to store the pid of sshd, and never again. As a result, the realconfig_perfect case has 1 more routine reported as suspect than the realconfig_imperfect cases.

The dynamic analysis identifies 157 routines whose behavior has deviated when going from version 4.1 to 4.2. Again, the number of deviations is higher than the number of modified routines, because the upgraded code fails much earlier than the original one.

The intersection of the two analyses results in only 6 routines (*do_child*, *do_exec*, *do_exec_no_pty*, *packet_start*, *session_setup_x11fwd*, and *session_close*) in the random case, and 7 (*do_authentication2* is the additional routine) in the realconfig cases. 3 of the 6 (or 7) primary suspect routines are key to understanding the failure. However, the single modification in the upgrade that directly causes the failure is in the session_setup_x11fwd routine.

**OpenSSH: ProxyCommand bug.** Recall that this bug affected ssh in version 4.9, which comprises 58K LOC and 1535 routines (712 routines in ssh). The upgrade to this version modified 122 routines.

Table 4.2 presents the results for each of the analyses in Sahara and all techniques combined for every experiment. The feature selection step results in 2 features (out of 354) chosen as suspect in the random cases. Both the features: the shell environment parameter `SHELL`, and the configuration parameter `Host` are related to the bug, though `SHELL` is the culprit environment resource. `Host` feature is related to the `ProxyCommand` option in *ssh_config*. Like before, feature selection for this bug is pretty accurate in the random experiments, due to the way we assigned values to the other environment resources. These two features correspond to 4 variables in the ssh code.

In case of realconfig cases, the feature selection results in 3 suspect features: the configuration parameter `HostKeyAlias`, in addition to `SHELL` and `Host` features. Again, `HostKeyAlias` corresponds to the `ProxyCommand` option in the ssh_config.

The static analysis produces 10 suspect routines for the random experiments, and 15 suspect routines for the realconfig experiments. The 10 suspect routines comprise those that: (1) issue and setup up a ssh session (*main, client_loop, ssh_session, ssh_session2, and ssh_session2_open*); (2) setup the proxy and execute a local command (*ssh_proxy_connect and ssh_local_cmd*); and (3) copy the command into and out of the buffer (*pwcopy, strlcpy, and xstrdup*). In the realconfig cases, all the 10 routines described above are suspect, in addition to those affected by `HostKeyAlias` (*check_host_key, put_host_port, ssh_connect, ssh_login, and verify_host_key_dns*).

The dynamic analysis identifies 284 routines whose behavior has deviated between versions 4.7 and 4.9. The intersection of *SuspectedRoutines* and *DeviatedRoutines* is only 7 routines in the random cases and 11 routines in the realconfig cases. The 4 routines pertaining to the session setup get filtered out as their behavior did not change after the upgrade.

From these results, we can see that the number of primary suspects found by Sahara is at least 1.4x lower than when using static analysis alone, at least 14x lower than when using dynamic analysis alone, and 15x lower than the number of routines that were actually modified. Again, these results illustrate Sahara's ability to focus the debugging of failed upgrades on a small number of routines, even when many users do not experience failures despite having environment resources that could trigger bugs in the upgrade.

**Impact of number of profiles with failure-inducing settings.** So far, we have studied the impact of imperfections in the categorization of success/failure of the upgrades on the behavior of Sahara. Another key factor for the effectiveness of feature selection is the percentage of user profiles that actually include the environment resource settings that cause the upgrade failures. On one hand, the lower this percentage, the less information we have about the failures and, thus, the worse the feature selection results should be. On the other hand, lowering this percentage reduces noise (i.e., supporting evidence for resources that are not related to the failures) in the dataset

| Bug | Experiment | SERs (feature selection) | SRs (static analysis) | DRs (dynamic analysis) | Primary Suspects (Sahara) |
|---|---|---|---|---|---|
| Port | random_perfect30 | 1 | 12 | 124 | 6 |
| | random_perfect | 1 | 12 | 124 | 6 |
| | random_perfect10 | 1 | 12 | 124 | 6 |
| | realconfig_perfect30 | 1 | 12 | 124 | 6 |
| | realconfig_perfect | 3 | 22 | 124 | 7 |
| | realconfig_perfect10 | 3 | 22 | 124 | 7 |
| X11 | random_perfect30 | 1 | 18 | 157 | 6 |
| | random_perfect | 1 | 18 | 157 | 6 |
| | random_perfect10 | 1 | 18 | 157 | 6 |
| | realconfig_perfect30 | 1 | 18 | 157 | 6 |
| | realconfig_perfect | 3 | 21 | 157 | 7 |
| | realconfig_perfect10 | 2 | 20 | 157 | 6 |
| Proxy | random_perfect30 | 2 | 10 | 284 | 7 |
| | random_perfect | 2 | 10 | 284 | 7 |
| | random_perfect10 | 2 | 10 | 284 | 7 |
| | realconfig_perfect30 | 3 | 15 | 284 | 11 |
| | realconfig_perfect | 3 | 15 | 284 | 11 |
| | realconfig_perfect10 | 5 | 29 | 284 | 21 |

Table 4.3: Impact of number of profiles with failure-inducing settings (Port = Port Forwarding; X11 = X11 forwarding; Proxy = ProxyCommand; SRs = Suspect Routines; DRs = Deviated Routines).

and may lead to better selection results. To confirm these observations, we performed some experiments in which we varied the number of such profiles. In particular, we considered cases in which 30 or 10 profiles (out of 87) had the failure-inducing settings. Recall that our default results above assumed 20 such profiles.

Table 4.3 presents the "perfect" results from these experiments. The default results (random_perfect and realconfig_perfect) and the dynamic analysis results are included for clarity. As expected, the number of SERs (as well as suspect routines and primary suspects) tends to increase when we lower the number of profiles with failure-inducing settings. Interestingly, the realconfig results for the X11 forwarding bug show that lowering noise (going from realconfig_perfect to realconfig_perfect10) can indeed improve results as well.

**Impact of feature selection accuracy.** Feature selection is a major component of Sahara in that it defines the scope of the static analysis. Recall that Sahara's feature selection considers all the features that are within 30% of the highest ranked feature as SERs by default. Here, we study two additional scenarios: (1) all features that are within 50% of the highest ranked feature are considered SERs, and (2) all OpenSSH configuration parameters are considered SERs. These scenarios cause an increasing number of unnecessary SERs.

For the port forwarding bug and scenario (1), the number of SERs remains the same in all the *random* cases and the *realconfig_perfect* case. In the *realconfig_imperfect60* case, the SERs increase from 3 to 4 and the prime suspects from 7 to 14. In the *realconfig_imperfect20* case, the SERs increase from 3 to 6 and the prime suspects from 7 to 18. In scenario (2), the number of SERs is 22 (all ssh parameters) and the number of prime suspects is 34.

For the X11 forwarding bug and scenario (1), the number of SERs remain the same in all the *random* cases. In the *realconfig_perfect* case, the SERs increase to 9 and the prime suspects to 10. In the *realconfig_imperfect60* case, the SERs increase to 11 and the prime suspects to 10, whereas in the *realconfig_imperfect20* case, the SERs increase to 12 and the prime suspects to 11. In scenario (2), the number of SERs increases to 51 (all sshd parameters) and the number of prime suspects to 43.

These results illustrate the behavior we expected: the less accurate feature selection is, the more prime suspects Sahara finds. Defining a few more SERs than necessary does not increase the number of prime suspects excessively (roughly by 2x at most, in comparison to our default results). However, adding too many unnecessary SERs can increase the number of prime suspects by 6x-7x, as in scenario (2).

**SQLite bug.** We injected this bug in SQLite version 3.6.14.2, which comprises 67K LOC and 1338 routines. The upgrade modified two routines. We ran only the random family of experiments, since this was not a real upgrade bug. These results show that feature selection identified 2-3 SERs, static analysis produced 12-13 *SuspectRoutines*, and dynamic analysis identified 14 *DeviatedRoutines*. Sahara outputs 2 primary suspects in each of the three random cases (exactly the routines that were modified); one of the prime suspects is the root cause of the failure. Again, although trivial, these experiments illustrate that Sahara can be used without modification for a variety of applications.

**uServer bug.** We injected this bug in uServer version 0.6.0, which comprises 37K LOC and 404 routines. The upgrade modified 10 routines. Again, we ran only the random family of experiments, since this was not a real upgrade bug. The experiments stopped at the feature selection step, since the ranks of the top-ranked features consistently exhibit high standard deviations. Thus, feature selection properly flags this bug as unrelated to the environment.

**Summary.** The Sahara results for the five bugs and the different imperfections we studied indicate that our system may significantly reduce the time and effort required to diagnose the root cause of upgrade failures.

# Chapter 5
# Related Work

## 5.1 Characterizing Upgrades

A few prior works have characterized upgrades [5, 12, 18, 48]. Beattie *et al.* [5] tried to determine the best time to apply security patches. They built mathematical models to determine when to patch, and validated their models using the empirical data. They focused solely on security patches and did not consider the reasons for the buggy patches. Gkantsidis et al. [18] focused on the Windows environment and on the networking aspects of deploying upgrades. They did not consider several important issues, including frequency of upgrade problems or their root causes. An inspection of 350,000 machines by Secunia [48] found that 28% of all major applications lack the latest security upgrades. However, they focused on security upgrades alone, and did not address upgrade problems. In contrast to all these studies, our characterization focuses on all types of problems reported of a single application (OpenSSH) extending multiple versions. Furthermore, we perform a more detailed characterization of the upgrade problems and their root causes.

The survey done by Crameri *et al.* is the closest in motivation to our work. They surveyed 50 system administrators to ascertain the frequency of upgrades and their problems, and characterize the reasons for the upgrade problems. However, their focus was much broader and therefore, shallower than ours. In this thesis, we do an in-depth analysis of the bugs reported across different versions of an application. We manually inspected 96 OpenSSH bug reports to determine the frequency of the problems caused by user's input and/or environment, frequency of the upgrade problems, and the types and frequency of various upgrade problems.

## 5.2 Upgrade Deployment and Testing

A few studies [12, 33, 34] have proposed automated upgrade deployment and testing techniques. McCamant and Ernst [33,34] automatically identify incompatibilities when upgrading a component in a multi-component system. However, neither of these works attempted to prevent the incompatibilities. Crameri *et al.* [12] proposed deploying upgrades in stages to clusters of users that have similar environments. However, none of the studies attempted to identify the aspects of the environment that are correlated with the failure, considered user's past execution behavior, or tried to isolate the root cause of the failures at the users' sites.

## 5.3 Recommendation Systems

Prior research [22, 47, 49] has used collaborative filtering to recommend videos, articles and music based on the preferences of other users with similar tastes. They build on the principle that past similarity between users is a good predictor of the user's future behavior (choices). Mojave employs the principles of collaborative filtering to build the first upgrade recommendation system. Specifically, Mojave uses similarity between a new user's environment settings and past program execution behavior with other users where the upgrade failed (or succeeded) to recommend for or against an upgrade.

## 5.4 Automated Debugging

**Classification of failure reports.** [13, 42] grouped failure reports using machine learning and call sequence similarity to aid the diagnosis and debugging of software failures. [42] applied supervised and unsupervised learning to assess the frequency and severity of failures caused by particular defects, and to help with diagnosis of those defects. The approach samples execution profiles through the feedback from the instrumented code running at user sites. In addition, it uses a profiling tool to identify the features (aspects of the execution) that are significant to the reported failures. Then, it runs feature selection to identify a subset of features that are most useful in grouping failures, and use this subset to group the bug reports. [13] used average levenshtein distance between two stack traces to measure similarity. The approach analyzed crash reports to group reports such that each class has reports from only one bug (could result in faster bug resolution).

Mojave also relies on call sequence (dis)similarity, along with users' environment settings, and leverages classification to learn a failure prediction model. Specifically, Mojave uses $90^{th}$ percentile length of LCS between (success or failed) call sequences to measure similarity between the program behavior at different user sites. Furthermore, Mojave performs feature selection on the user attributes to identify the aspects of the user site that are the most likely causes of the failures. However, Mojave seeks to prevent upgrade failures for new users, rather than isolate the root cause of the failure.

**Troubleshooting mis-configurations.** The idea of Snitch [35] and PeerPressure [52] is to identify the root cause of software mis-configurations using machine learning techniques. PeerPressure performs statistical analysis of Windows registry snapshots from a large number of machines. After a misconfiguration is detected, PeerPressure re-executes the program in a special tracing environment to capture the relevant registry data. It then uses Bayesian estimation to compare each mis-configured machine's registry values with those of the machines that can successfully run the same program. Rare registry values that correlate well with mis-configurations are coerced to the more common values. Snitch introduces Interactive Decision Trees (IDT) to allow the developer to guide the troubleshooting process, starting from configuration traces from many users.

ConfAid [3] helps debug mis-configurations without information from other users. Instead, it instruments the binaries to track the causal dependencies between application-level configuration parameters and output behavior. The binaries, parameters, and outputs of interest are specified manually.

These three systems assume that the software is correct, but was mis-configured by its users. Mojave and Sahara are fundamentally different. Mojave seeks to prevent upgrade failures for future users by isolating the failure characteristics and leveraging (dis)similarity between the new user and the existing users. Mojave uses machine learning to determine if an upgrade failure is environment-related, and to learn the prediction model. Sahara aims to help find upgrade bugs that are triggered by proper configurations and environments. Moreover, both Mojave and Sahara go well beyond finding the environment resources most likely to be related to a bug (i.e., feature selection).

Qin *et al.* [44] observe that many bugs are correlated with the "execution environment" (which they define to include configurations and the behavior of the operating and runtime systems). Based on this observation, they propose Rx, a system that tries to survive bugs at run time by dynamically changing the execution environment. A follow-up to Rx, Triage [51] goes further by dynamically changing the execution environment while attempting to diagnose failures at users' sites.

In this work, Mojave seeks to prevent upgrade bugs or misbehavior for new users before they install the upgrade, rather than the bugs that appear much after they have applied the upgrade. In addition, Sahara focuses on simplifying debugging of upgrade bugs or misbehavior, rather than software bugs in general as Rx and Triage do. For this reason, Sahara can be much more specific about which variables and routines should be considered first during debugging. Moreover, Sahara can handle bugs due to aspects of the environment that would be difficult (or impossible) to change without semantic knowledge of the application. Finally, Rx and Triage do not leverage data from many users, machine learning, or static analysis. Using any of these features could speed up Triage's diagnosis. In fact, as we argue in Section 4.4, Sahara is complementary to systems like Triage.

**Diagnosis using execution profile mining.** Several previous papers [6,14,15,31,36, 55] have employed data mining and machine learning techniques on execution profiles for problem diagnosis. [31, 55] used graph mining, feature selection, and classification algorithms to localize non-crash bugs. Specifically, they analyzed the differences among the graphs of success and failed executions to identify the subgraphs that are correlated with the failures. They used these subgraphs to determine if a given graph is a success or failed execution, and to localize the root cause of the problem. In contrast, Mojave compares the environment settings and past execution sequences of the new user with other users that have installed the upgrade to predict if an upgrade will misbehave for him or her. Mojave aims to prevent failures (crashes and non-crashes) for the new user. Sahara isolates a small subset of routines that are likely to be the root cause of the upgrade failures by leveraging the user's environment settings, and a novel combination of machine learning and program analyses.

Dickinson *et al.* [14] used cluster analysis of execution profiles to find failures among the executions induced by a set of test cases. Specifically, the analysis used the dissimilarity or the peculiarity of a few executions as an indication of a failed execution. They attempted to label the execution profiles as success or failure using clustering. In our work, we use similarity between a user's past execution profile with that of other users where the upgrade succeeded (or failed) to predict if the upgrade would fail for this user. Furthermore, we simplify debugging of environment-related upgrade failures.

In [36], Mirgorodskiy *et al.* used lightweight dynamic instrumentation to collect function call traces from software running at user sites. They compared the call traces, and run classification to isolate the subset of the trace or a single function that is the root cause of the failure. Mojave uses call sequences as program behavior representatives too. However, it computes similarity between users' call sequences and environment settings to predict if the upgrade is likely to succeed or fail, not isolate the root cause of the failure. In addition, in Mojave the instrumented code runs briefly at the time of upgrade to collect data.

**Statistical debugging with user feedback.** Several previous works [2, 11, 19, 28–30, 42, 58] relied on low-overhead, privacy-preserving instrumentation infrastructures to provide user execution data back to developers. For example, Cooperative Bug Isolation (CBI) [27] constitutes a feedback loop between developers and users. Developers provide instrumented software to users, and users provide data about that software's behavior in their environments. The instrumentation consists of predicates placed at different points of the program. Developers then use sophisticated statistical and regression algorithms to rank predicates based on how well they correlate to bugs. To reduce the manual work, [24] extended CBI to find the control flow paths connecting the highly ranked predicates. In our work, we rely on information gathered at user sites, but the data collection lasts temporarily after the upgrade for initial users, and only before the upgrade for the new users. We employ regression algorithms to identify user attributes that are highly correlated with the upgrade misbehavior. Mojave goes further by leveraging those attributes to predict and prevent similar misbehavior for new users.

Arnold *et al.* [2] sampled stack traces over a period of time, and generated a call graph prefix tree to assemble a profile of the application's behavior. The prefix tree enabled comparison of stack traces to identify anomalies between stack traces from different sites, and locate errors. Unlike this work, Mojave compares past execution call sequences (along with environment settings) to predict if an upgrade is likely to misbehave for a user; not locate errors or identify anomalies. Furthermore, Sahara compares call sequences from two versions of the program at a single user site to reduce the scope of debugging.

**Static analysis.** Several researchers have used static analysis for debugging, e.g. [16, 37, 40]. In [16], authors used static analysis to infer key program properties, and then check the system against these beliefs to uncover bugs. [37] used the rate of past failure occurrences and complexity of a software component as predictors for future failures. In patchAdvisor [40], authors combine static analysis of control-flow graphs with dynamic execution traces to study the potential impact of a patch in the field. However, none of these approaches considered the impact of the users' environment and/or inputs on the upgrade failures. In addition, our use of static analysis differs from these approaches: we do not use it to find the bugs themselves; rather, we use it to reduce the length of the call sequences for environment-related bugs (resulting in speedier similarity computation) in case of Mojave, and to reduce the scope of debugging in case of Sahara.

**Dynamic analysis.** Some studies [6, 17, 20] automatically extracted likely program invariants based on dynamic program behavior (possibly after running multiple times with different inputs to increase coverage). The detection of invariants may involve significant overhead. Software can be deployed to users with instrumentation to check the invariants. Developers can then use the invariants and any violations of them to aid in debugging, just as the predicates above can be used. Our work is different from theirs in the following ways: 1) We consider user environment as a failure predictor and they do not; 2) We use the commonality between execution profiles (or the lack thereof) as a failure predictor rather than the invariants over the executions; 3) We use the learned prediction model and execution similarity to prevent upgrade failures for future users; and 4) We restrict the execution of instrumented versions of the software to a very short time (just before or briefly after the upgrade).

A few prior works [11, 23, 58] have used delta debugging to resolve regression faults automatically and effectively. They focused on comparing program states of failed and successful runs to identify the space of variables or rank program statements that are correlated with the failure. In this thesis, we use dynamic analysis to compute similarity between users, or to compute the difference between two runs of a program. However, our approach is driven by environment resources and combines information from a collection of users, machine learning, static analysis, and dynamic analysis. Furthermore, unlike delta debugging, we neither require instrumenting the production code nor replaying the execution multiple times at the users' sites.

Xie and Notkin [54] proposed program spectra to compare versions and get insights into their internal behavior. Harrold *et al.* [21] found that the deviations between spectra of two versions frequently correlate with regression faults. Sahara uses value spectra to compare the execution call traces from before and after the upgrade is applied. However, merely identifying the deviations in the upgraded version leads to a large number of candidates for exploration, as our experiments demonstrate. The same is likely to occur for most large applications or major upgrades. Sahara further narrows down the deviation sources by cross-referencing them with suspect routines found through information from users, machine learning, and static analysis.

The aim of [39, 57] is to detect the root cause of regression failures automatically. Ness and Ngo [39] used a linear search algorithm on the fully-ordered source management archive to identify a single failure-inducing change. In [57], the authors proposed an algorithm to determine the minimal set of failure-inducing changes. These studies sought to isolate the fault-inducing change after a regression test fails at the developer's site. In contrast, Sahara assumes that the upgrade has been tested thoroughly at the developer's site and is deployed after all tests have passed. Sahara helps isolate the fault-inducing code that is affected by specific user environments. These failures are not easily reproducible at the developer's site because of environmental differences.

**Other approaches.** Researchers have actively been considering other approaches to automated debugging, such as static analysis, model checking, and symbolic execution, e.g. [9, 16, 56]. Our work is not closely related to any of these approaches, except peripherally for our use of static def-use analysis.

# Chapter 6
# Conclusion

In this dissertation, we sought to simplify the management of software upgrades by preventing upgrade failures for new users, and guiding the debugging of failed software upgrades.

We motivated our work using a survey of 96 OpenSSH bugs spanning five versions. The results confirm that upgrade failures are frequent, and that the majority of the upgrade bugs are caused by the users' environment settings and/or their inputs. We argue that the developer and the users could collaborate to prevent most of these upgrade problems, and considerably reduce the debugging effort. To this end, we introduced two systems that leverage environment information and dynamic execution data from many users to prevent upgrade failures and simplify their debugging.

Mojave, the first upgrade recommendation system, recommends in favor or against an upgrade to new users. Driven by the fact that most upgrade failures are environment- and/or input-related, Mojave gathers existing users' environment, dynamic execution data and success/failure flags. Mojave then combines machine learning, and dynamic and static source analyses to identify the user attributes that are highly correlated with the failure, compares them to the new users' attributes to predict whether the upgrade would succeed or fail for them. Our evaluation with five upgrade failures across three applications demonstrates that Mojave provides accurate recommendations to the majority of users.

Sahara, the upgrade debugging system, reduces the effort developers must spend to debug failed upgrades by prioritizing the set of routines to consider when debugging. Given that most upgrade failures result from differences between the developers' and users' environments, Sahara combines information from user site executions and environments, machine learning, and static and dynamic analyses. We evaluated Sahara for five bugs in three widely used applications. Our results showed that Sahara produces accurate and a small set of prime suspect routines. Importantly, the set of recommended routines remains small and accurate, even when the user site information is misleading or limited.

In conclusion, our results demonstrate that combining user feedback, machine learning, and dynamic and source analyses can prevent most of the upgrade failures for new users, and their debugging can be largely simplified.

Looking to the future, we expect that this particular combination of techniques can become even more useful in improving software quality. In particular, an increasing number of users are willing to provide extensive information about their interests and preferences. Internet services have used this information for service personalization and performance improvement, both of which require machine learning. These users may also be willing to provide information about their systems and many aspects of their experience with and use of software. This wealth of information will be invaluable to future developers.

# References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Practices and Techniques*. Addison-Wesley, 1986.

[2] D. Arnold, D. Ahn, B. Supinski, G. Lee, B. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In *Proceedings of the Symposium on International Parallel and Distributed Processing*, 2007.

[3] M. Attariyan and J. Flinn. Automating Configuration Troubleshooting With Dynamic Information Flow Analysis. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2010.

[4] R. Bachwani, O. Crameri, R. Bianchini, D. Kostić, and W. Zwaenepoel. Sahara: Guiding the debugging of failed software upgrades. In *Proceedings of IEEE International Conference on Software Maintenance*, 2011.

[5] S. Beattie, S. Arnold, C. Cowan, P. Wagle, and C. Wright. Timing the application of security patches for optimal uptime. In *Proceedings of the Large Installation System Administration Conference*, 2002.

[6] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the International Conference on Software Engineering*, 2004.

[7] X forwarding will not start when a command is executed in background. https://bugzilla.mindrot.org/show_bug.cgi?id=1086.

[8] Connection aborted on large data -R transfer. https://bugzilla.mindrot.org/-show_bug.cgi?id=1360.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the International Symposium on Operating Systems Design and Implementation*, 2008.

[10] A. Chandra, D. Mosberger, and Linux Performance. Scalability of linux event-dispatch mechanisms. In *Proceedings of the USENIX Annual Technical Conference*, 2001.

[11] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of International conference on Software engineering*, 2005.

[12] O. Crameri, N. Knezevic, D. Kostić, R. Bianchini, and W. Zwaenepoel. Staged deployment in mirage, an integrated software upgrade testing and distribution system. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2007.

[13] T. Dhaliwal, F. Khomh, and Y. Zou. Classifying field crash reports for fixing bugs: A case study of mozilla firefox. In *Proceedings of the International Conference on Software Maintenance*, 2006.

[14] William Dickinson, David Leon, and Andy Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the International Conference on Software engineering*, 2001.

[15] F. Eichinger, K. Böhm, and M. Huber. Mining edge-weighted call graphs to localise software bugs. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, 2008.

[16] D. Engler et al. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the International Symposium on Operating Systems Principles*, 2001.

[17] M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of International conference on Software engineering*, 1999.

[18] C. Gkantsidis, T. Karagiannis, P. Rodriguez, and M. Vojnović. Planet scale software updates. In *Proceedings of the ACM Conference on Communications Architectures and Protocols*, 2006.

[19] K. Glerum et al. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of Symposium on Operating Systems Principles*, 2009.

[20] S. Hangal and M. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of International conference on Software engineering*, 2002.

[21] M. J. Harrold, Y. G. Rothermel, Z. K. Sayre, Z. R. Wu, and L. Y. Z. An empirical investigation of the relationship between spectra differences and regression faults. *Journal of Software Testing, Verification and Reliability*, 2000.

[22] W. Hill, L. Stead, M. Rosenstein, and G. Furnas. Recommending and evaluating choices in a virtual community of use. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, 1995.

[23] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2008.

[24] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2007.

[25] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the International Joint Conference on Artificial Intellligence*, 1995.

[26] N. Landwehr, M. Hall, and E. Frank. Logistic model trees. In *Machine Learning*, 2003.

[27] B. Liblit. *Cooperative Bug Isolation.* PhD thesis, University of California, Berkeley, 2004.

[28] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, 2005.

[29] B. Liblit et al. Bug isolation via remote program sampling. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, 2003.

[30] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. Sober: Statistical model-based bug localization. *Proceedings of European Software Engineering conference held jointly with the ACM Symposium on Foundations of software Engineering*, 2005.

[31] C. Liu, X. Yan, H. Yu, J. Han, and P. Yu. Mining behavior graphs for backtrace of noncrashing bugs, 2005.

[32] Z. Markov and I. Russell. An introduction to the weka data mining system. In *Proceedings of Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 2006.

[33] S. Mccamant and M. Ernst. Predicting problems caused by component upgrades. In *Proceedings of European Software Engineering conference held jointly with the ACM Symposium on Foundations of Software Engineering*, 2003.

[34] S. Mccamant and M. Ernst. Early identification of incompatibilities in multi-component upgrades. In *Proceedings of the European Conference on Object-Oriented Programming*, 2004.

[35] J. Mickens, M. Szummer, and D. Narayanan. Snitch: interactive decision trees for troubleshooting misconfigurations. In *Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*, 2007.

[36] A. Mirgorodskiy, N. Maruyama, and B. Miller. Problem diagnosis in large-scale computing environments. In *Proceedings of the Conference on Supercomputing*, 2006.

[37] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the International Conference on Software engineering*, 2006.

[38] G. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the International Conference on Compiler Construction*, 2002.

[39] B. Ness and V. Ngo. Regression containment through source change isolation. In *Proceedings of International Computer Software and Applications Conference*, 1997.

[40] J. Oberheide, E. Cooke, and F. Jahanian. If it ain't broke, don't fix it: challenges and new directions for inferring the impact of software patches. In *Proceedings of the 12th conference on Hot topics in operating systems*, 2009.

[41] OpenSSH release dates. http://openbsd.mirrors.hoobly.com/OpenSSH/portable.

[42] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.

[43] ProxyCommand not working if $SHELL not defined. http://marc.info/?l=openssh-unix-dev&m=125268210501780&w=2.

[44] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2005.

[45] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1986.

[46] M. O. Rabin. Fingerprinting by random polynomials. *Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University*, 1981.

[47] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: An open architecture for collaborative filtering of netnews. In *Proceedings of the ACM conference on Computer supported cooperative work*, 1994.

[48] Secunia "Security Watchdog" Blog. http://secunia.com/blog/11.

[49] U. Shardanand and M. Pattie. Social information filtering: Algorithms for automating "word of mouth". In *Proceedings of the conference on Human factors in computing systems*, 1995.

[50] SQLite home page. http://www.sqlite.org/.

[51] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2007.

[52] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[53] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[54] T. Xie and D. Notkin. Checking inside the black box: Regression testing based on value spectra differences. In *Proceedings of IEEE International Conference on Software Maintenance*, 2004.

[55] C. Yuan, N. Lao, J. Wen, J. Li, Z. Zhang, Y. Wang, and W. Ma. Automated known problem diagnosis with event traces. In *Proceedings of the European Conference on Computer Systems 2006*, 2006.

[56] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of Eurosys*, 2010.

[57] A. Zeller. Yesterday, my program worked. today it does not. why? In *Proceedings of European Software Engineering conference held jointly with the ACM Symposium on Foundations of Software Engineering*, 1999.

[58] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the ACM Symposium on Foundations of Software Engineering*, 2002.