Mojave: A Recommendation System for Software Upgrades *

Rekha Bachwani, Olivier Crameri†, Ricardo Bianchini, and Willy Zwaenepoel†
Rutgers University, USA †EPFL, Switzerland

Abstract

Software upgrades are frequent. Unfortunately, many of the upgrades either fail or misbehave. We argue that many of these failures can be avoided for new users of each upgrade by exploiting the characteristics of the upgrade and feedback from the users that have already installed it. To demonstrate that this can be achieved, we build Mojave, the first recommendation system for software upgrades. Mojave leverages data from the existing and new users, machine learning, and static and dynamic source analyses. For each new user, Mojave computes the likelihood that the upgrade will fail for him/her. Based on this value, Mojave recommends for or against the upgrade. We evaluate Mojave for two real upgrade problems with the OpenSSH suite. Initial results show that it provides accurate recommendations.

1. Introduction

Modern software is complex and consists of many interrelated components. Due to this complexity, developers frequently have to produce upgrades to fix bugs, patch security vulnerabilities, add or remove features, and perform other critical tasks. It is infeasible, if not impossible, for the developers to envisage, much less test their upgrades with every possible environment setting and input with which users drive the software. As a result, many of the upgrades either fail or misbehave for some users. We define *upgrade failures/bugs* as misbehavior caused by the code introduced in an upgrade, not just crashes.

A survey by Crameri *et al.* [6] showed that 90% of system administrators upgraded their software at least once a month, and that 5-10% of these upgrades misbehaved. The same survey also found that the difference between the environment (i.e., version of the operating system and libraries, configuration settings, hardware, etc) at the developer's site and users' sites was a major source of upgrade failures.

To further study the prevalence of upgrade failures, and the reasons for these problems, we surveyed 96 bug reports about the OpenSSH suite. Our intention was to study around 100 bug reports spanning multiple consecutive versions of OpenSSH. The following set of five versions met this criterion: 4.1p1, 4.2p1, 4.3p1, 4.3p2, and 4.5p1. The results show that 41% of the problems reported were due to upgrades. Furthermore, we find that the majority of the bugs are caused by both the user's environment and input.

* This research was partially funded by NSF grant CNS-0916878.

To avoid some upgrade failures, 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. Users try to avoid failures by delaying the installation of each upgrade until after many other users have installed it and provided positive feedback on it. Crameri showed that 70% of the administrators refrain from installing each upgrade at first. However, a positive feedback from others may be useless if their environments and/or inputs are different.

Obviously, deploying upgrades as packages and/or delaying upgrades is not enough to prevent upgrade failures. Instead, we argue that the developer and the users 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" (users who intend to install the upgrade).

Along these lines, this paper proposes Mojave, the first recommendation system for software upgrades. Mojave's design is based on 4 observations: (1) upgrade bugs often take significant time to fix, so delaying crucial upgrades for users that may not be affected by the bugs is unnecessary and inefficient; (2) the upgrade failures are most likely caused by the particular characteristics of the corresponding users' environments or inputs; (3) new users that have characteristics similar to those of the existing users where the upgrade has failed are likely to experience similar failures; and (4) two users' sites at which the software behaved similarly in the past (execution behavior prior to the upgrade) are 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 [12, 25, 26].

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 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.

We evaluate Mojave on 2 real upgrade problems with OpenSSH. Mojave accurately predicts the upgrade failure for $96 - 100\%$ of the users. Given the accuracy of the

```
1.  #define PSHELL "/bin/sh"
2.  int env2 = 0;
3.
4.  int checklength(int len) {
5.    if (len <= 9)
6.      return len;
7.    else
8.      return -1;
9.  }
10. void do_option(int x) {
11.   printf("\nGot option %d",x);
12. }
13. int main(int argc, char *argv[]) {
14.   int retval1 = 0, i = 1;
15.   char shell[80];
16.   if (argc >= 2) {
17.     while (argv[i] != NULL) {
18.       if (strcmp(argv[i],"Option1") == 0) {
19.         do_option(1);
20.       }//end strcmp
21.       if (strcmp(argv[i],"Proxycmd") == 0) {
22.         strcpy(shell,PSHELL);
23.         env2 = strlen(shell);
24.         retval1 = checklength(env2);
25.         if (retval1 > 0)
26.           printf("\nSuccess:proxycmd");
27.         else
28.           printf("\nOops:checklength failed");
29.       }//end strcmp
30.       i++;
31.     }//end while
32.   }//end argc
33. }//end main
```

Figure 1. Example (failure-inducing) upgrade.

preliminary results, we expect that Mojave would be able to prevent most upgrade failures in the field.

2. Mojave

2.1 Motivating Example

Consider the example in Figure 1. It reads 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, and -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.

```
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`.

Although 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`.

Next, we describe Mojave's learning and recommendation phases.

2.2 Learning Phase

Upgrade deployment, tracing, and user feedback (steps 0-3). As shown in Figure 2, Mojave first 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).

Mojave uses the tracing infrastructure detailed in Mirage and Sahara [2, 6] to record the inputs and identify all the "environmental resources" a software depends on. The environmental resources include the name and version of the operating system and libraries, the configuration settings, the name and version of the other software packages installed, and a description of the hardware. Mojave allows the developer to include or exclude additional resources. In addition, Mojave leverages the parsers created for Mirage and Sahara to compute a concise representation (key/value fingerprint) for each environmental resource. The key of each fingerprint is the name of the resource, and the value is its hash value. The user input includes any command line arguments and data manually entered as input.

A call sequence comprises the routines executed during one run of the software. Mojave may collect multiple call sequences from each user. To collect call sequence data, Mojave uses the *C Intermediate Language* (CIL) [21] to automatically instrument the application. A new CIL module, called *call-logger*, inserts calls to our runtime library that logs the names of the routines executed. In case the software forks multiple processes, the module logs the call sequence for each process individually.

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 obscures and then sends this information, along with the environment and call sequence data, back to the developer's site (step 3). 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. Mojave collects these data from all users that are willing to participate. User profiles from all sites serve as the input to the other steps. Now suppose that the upgrade misbehaved at one user site at least.

Call sequence filtering for environment-related failures (steps 4-6). With the users' environment and upgrade success/failure information, Mojave runs a machine learning algorithm to determine if the misbehavior is likely due to

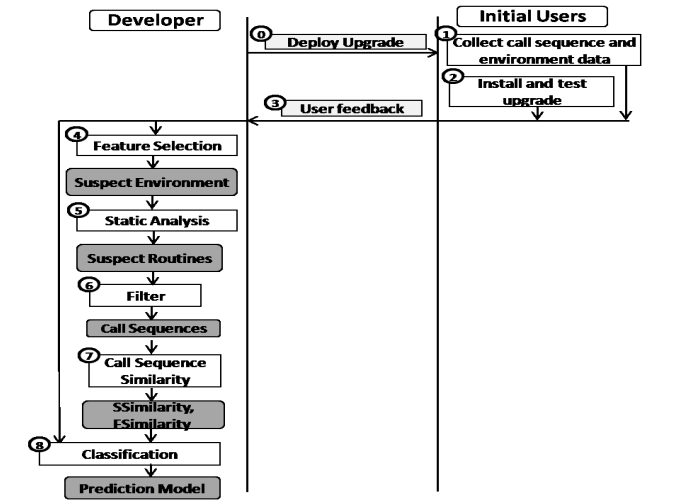

Figure 2. Learning phase in Mojave.

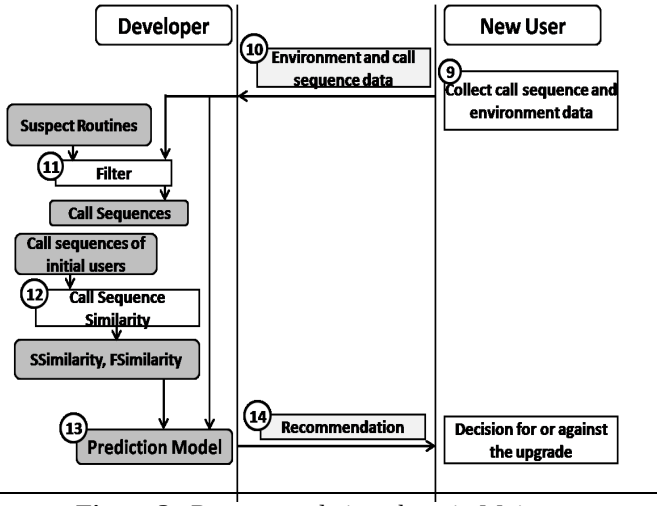

Figure 3. Recommendation phase in Mojave.

some aspect of the environment (step 4). For this, it uses Sahara's feature selection and static analysis infrastructure [2]. Specifically, Mojave uses the GainRatio [24] decision tree algorithm with feature ranking [16] for selection. The algorithm builds a decision tree by recursively selecting a feature that splits up the dataset into 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 Figure 1, the root feature would be the SHELL variable. The subsets with SHELL strings of length 0 are failures, whereas the others are successes.

The output of the algorithm is a set of features, their Gain Ratios, and their ranks. To validate the feature selection and compute the standard deviation of the feature ranks, Mojave uses 10-fold cross-validation [13]. If the standard deviations of the top-ranked features are high, the environment is unlikely to be the reason for the failures. In this case, Mojave skips to the call sequence similarity step (step 7). Otherwise, 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.

Next, using def-use static analysis, Mojave isolates the variables in the pre-upgrade code that derive from those suspects; the routines that use these variables are considered suspect (step 5). Mojave performs static analysis using two CIL [21] modules, *call-graph* and *def-use*. The call-graph module computes a whole-program static call graph by traversing the source files. Every node in the call graph is a routine, and its children nodes are the routines it calls. The def-use module links all the variables that derive directly or indirectly from each SER using a def-use chain [1]. The module handles arrays as single variables, whereas the struct and union fields are handled separately.

Mojave analyzes each statement of a routine, starting with the root routine. For every variable access, it checks whether the variable is a suspect or is affected by another suspect, either directly or indirectly. If so, the variable and the corresponding routine become suspects. If a routine calls another routine with a suspect variable as an argument, the caller and the corresponding formal parameter of the callee become suspect. The callee becomes a suspect if the suspect parameter is used in the function, and not otherwise. Furthermore, a routine becomes suspect if the return value of any of its callees is suspect, and it is used in the routine. Similarly, a routine becomes suspect if any parameter passed by reference to one of its callees becomes suspect, and it is used in the routine. This step outputs the *SuspectRoutines* set, after the entire graph has been traversed. In Figure 1, `main` and `checklength` are the two suspect routines.

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, resulting in shorter sequences, and faster similarity computation (step 6). This step updates the call sequence data for all 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 4 illustrates the possible call sequences for the example in Figure 1. Figure 4(a) shows the call sequence when the program is run without the input arguments "Option1"

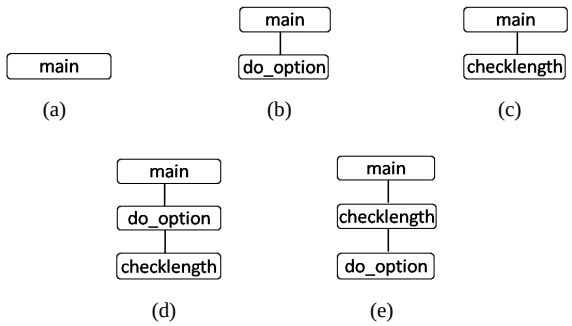

Figure 4. Call sequences for the example program.

and "ProxyCommand". Figures 4(b) and 4(c) depict the call sequence when the program is executed with "Option1" or "ProxyCommand" arguments, but not both. Figures 4(d) and 4(e) exhibit the call sequence when the program is executed with both arguments.

The LCS between the call sequences in Figure 4(a) and Figure 4(b), and between Figure 4(a) and 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 4(a) and Figure 4(d), and between Figure 4(a) and 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%. *FSimilarity* and *SSimilarity* range from $33-100\%$ for the example depending on the total number of initial users, the number of users where the upgrade passed or failed, and their call sequences.

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

Classification (step 8). This step takes the user profiles and tries to learn a binary classifier that gives good predictions on the test set. Specifically, Mojave uses the *Logistic Regression* [14] classification algorithm. Logistic Regression produces 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. 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 model for the example in Figure 1 is $p(fail) = a_0 + a_1 * (SHELL == Hash)$, where a_i are the weights that are learned from the training set, and $Hash$ is the hash of the failure-inducing shell name (empty string). If the value of $p(fail)$ is greater than 0.5, 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.

As we mention above, Mojave performs static analysis and call sequence filtering on the pre-upgrade version of the software. Obviously, this approach does not help identify the root cause of upgrade failures. Fortunately, Mojave's goal is different: it simply seeks to determine whether an upgrade is likely to fail at each new user site. *It relies on the fact that pre-upgrade behaviors are a good indication for post-upgrade success and failure.*

2.3 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).

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

Filtering for environment-related failures (step 11). If the upgrade is likely to have environment-related bugs, Mojave filters the new user's call sequence data with *SuspectRoutines* (from 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 (from the learning phase) to compute the probability that the new user belongs to 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, and not otherwise.

3. Evaluation

We now describe our methodology and results with 2 real bugs in OpenSSH.

3.1 Methodology

OpenSSH: Port forwarding bug. Port forwarding allows tunneling of TCP traffic over a secure shell. The bug [5] manifested for users that issued large transfers when using port forwarding in version 4.7. Some users observed the following error:

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

The transfers aborted because: (a) the users had enabled port forwarding using the `Tunnel` parameter in the *ssh_config* file; (b) the increase in the default window size from 128KB to 2MB in version 4.7 of the *ssh* client code; (c) port forwarding code incorrectly advertising the default window size as the default packet size; and (d) *sshd* limiting the maximum packet size to 256KB. Given these characteristics, when users issued large transfers (over the *ssh* tunnel), some of the packets exceeded the 256KB limit causing the abort after the upgrade.

OpenSSH: X11 forwarding bug. This bug [4] is a regression bug in version 4.2p1. When the users tried to start X11 forwarding in the background, they observed:

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

This problem occurred because the developers modified the X11 forwarding code in *sshd* to fix channel leaks, including closing connections whose session had ended. When the forwarding process was started in the background, there was no session attached to it, causing an immediate exit.

User data collection. To simulate a real-world upgrade deployment to a large number of users with varied environment settings, we collected system-environment (e.g., operating system and library versions) and hardware data from 87 machines at our site across two clusters. The system environment is similar within a cluster, but different across clusters. In terms of hardware data, there are multiple classes of machines within and across the clusters. Machines are assigned different application-specific configurations and inputs. The space of inputs and system, hardware, and application environments results in diverse environment settings overall.

We used the methodology described in Section 2.2 to (1) automatically generate instrumented versions of OpenSSH; (2) identify their environmental resources; and (3) collect call sequence data and compute success and failure similarity measures. We use parsers described in [2] for environmental resources. When Mojave collects call sequences, the software runs around 2X slower. (We ran all experiments on 2.8-GHz Intel Pentium 4 machines with 512MB of RAM and the Ubuntu 8.04.4 Linux distribution.)

In addition, we downloaded 8 complete OpenSSH configuration files from the Web. For each bug, we modify 3 of these files to include the settings that activate the bug. Furthermore, we use 8 inputs, 3 of them would trigger the bugs if the suspect environment settings were present, and the other 5 would not. One of the 8 configuration files and 1 input are 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 do not. Some of the 67 profiles may have failure-inducing input, but not the environment settings that activate the bug.

To mimic the situation where some users have failure-inducing settings, but their inputs do not activate the bug, we perform three types of experiments: $perfect$, $imperfect60$, and $imperfect20$. 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 a 100% correlation between those resources and the failure.

In the two imperfect cases, the environment settings are the same as in the perfect 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 cause the feature selection to pick more SERs for the environment-related failures than in the perfect case.

In all experiments, the feature selection step considers the features ranked within 30% of the highest ranked feature as suspects. Across all experiments, this step takes $1-3$ seconds, and the static analysis step takes $82-100$ seconds.

Learning and recommendation. Our experiments use two-thirds (57) of the profiles as training data to learn the prediction model, and the remaining one-third (30) as test data for the recommendation phase. We assume that users will install the upgrade irrespective of the recommendation, and report back if it succeeded or failed. In all experiments, this step takes $27-59$ seconds to learn the prediction model.

3.2 Results

OpenSSH: Port forwarding bug. Mojave identifies 101 environmental resources, many of which are split into smaller chunks; for others, each parameter is a separate feature. Overall, there are 325 features, forming the input to the feature selection. Feature selection ranks 3 features (configuration parameters) highly across all experiments. These 3 parameters correspond to 8 suspect variables in ssh. The static analysis results in 22 suspect routines.

Filtering shortens the sequences to 275–605 calls (from 6K–47K) for the users where the upgrade succeeded, and 380–632 calls (from 29K–73K) for the users where the up-

Bug	Experiment	Training		Test		Mojave Accuracy					
		Success	Failure	Success	Failure	True Pos.	True Neg.	False Pos.	False Neg.	Precision	Recall
Port	perfect	42	15	25	5	25	5	0	0	1	1
	imperfect60	48	9	27	3	27	2	0	1	1	0.96
	imperfect20	34	3	29	1	29	1	0	0	1	1
X11	perfect	42	15	25	5	25	5	0	0	1	1
	imperfect60	48	9	27	3	27	3	0	0	1	1
	imperfect20	34	3	29	1	29	1	0	0	1	1

Table 1. Recommendations for two bugs (Port = Port forwarding; X11 = X11 forwarding);

grade failed. This reduction speeds up the similarity computation significantly. The success and failure similarity measures are 79–100% and 80–98%, respectively. Mojave updates the user profiles with these similarities and passes 57 of the updated profiles to the classification algorithm. The classification outputs a prediction model comprising 1 feature for the $perfect$ case, 7 for the $imperfect60$ case, and 5 for the $imperfect20$ case.

Using the prediction model, Mojave computes recommendations for the 30 test profiles. Table 1 presents the results for the 3 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% accuracy (precision and recall of 1). In the $imperfect60$ case, it correctly predicts outcomes for all but 1 user, an accuracy of 97% (recall of 0.96).

OpenSSH: X11 forwarding bug. Mojave identifies 123 environmental resources, resulting in 354 features. Feature selection selects 3 features for the $perfect$ case. For the $imperfect60$ and $imperfect20$ cases, it also selects 3 features, 2 of which are the same as for the $perfect$ case. These 4 features correspond to 7 variables in sshd. Static analysis finds 20 suspect routines in the $perfect$ case, and 21 in the $imperfect$ cases. Filtering shortens the sequences to 104–107 calls (from 2.7K–81K) for the success, and 99–390 calls (from 2.8K–2.9K) for the failed instances. The success and failure similarities are 28–100% and 50–100%, respectively. The classification outputs a prediction model with 2 features for the $perfect$ case, 7 for the $imperfect60$, and 5 for the $imperfect20$ case.

Using the prediction model, Mojave computes recommendations for the 30 test profiles. It correctly predicts outcomes for all the new users, an accuracy of 100%.

Summary. Mojave provides recommendations with 96–100% accuracy (precision and recall in the 0.96–1 range), and can help prevent upgrade failures for most users.

4. Related Work

Recommendation systems. Prior research [12, 25, 26] has used collaborative filtering to recommend videos, articles, and music based on the preferences of other users with similar tastes. The principle is that past similarity between users is a good predictor of the user's future behavior. Mojave employs this principle to build the first upgrade recommendation system: it uses the similarity between (a) a new user's environment and past program execution behavior, and (b) those of users where the upgrade succeeded or failed.

Upgrade deployment, testing, and debugging. A few studies have focused on improving the management of upgrades [2, 6, 17, 18]. Crameri *et al.* [6] proposed deploying upgrades in stages to clusters of users that have similar environments. Bachwani *et al.* [2] collect information from many users to simplify the debugging of upgrades. Neither of these works attempt to predict future upgrade failures based on environment and past execution behavior. 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.

Machine learning and execution profiles in debugging. Without a focus on upgrades, previous work [7, 23] grouped failure reports using machine learning and call sequence similarity to aid the diagnosis and debugging of software failures. Other authors have used graph mining, feature selection, and classification algorithms on execution profiles to localize non-crash bugs [15, 30]. Dickinson *et al.* [8] used cluster analysis of execution profiles to find failures in a set of test executions. In [19], Mirgorodskiy *et al.* collected function call traces from software running at user sites. They compared the traces, and run classification to isolate the subset of the trace or a single function that is the root cause of the failure. Triage [28] dynamically changes the execution environment while attempting to diagnose failures at users' sites. PeerPressure [29] identifies the cause of misconfigurations by analyzing Windows registry snapshots from many machines. Autobash [27] uses an instrumented kernel and causal analysis to manage configurations.

Mojave is fundamentally different in that it prevents upgrade failures for future users, rather than finding the root cause of upgrade failures or troubleshooting misconfigurations. Nevertheless, its use of machine learning and execution profiles does have similarities to previous systems. However, Mojave is the first system to use these techniques on earlier versions of the software to predict behavior for later versions.

Source analyses in debugging. Several researchers have used static analysis for debugging (e.g., [9, 20, 22]). [20] used the rate of past failures and the complexity of software components as failure predictors. In patchAdvisor [22], the authors use static analysis of control-flow graphs to study the potential impact of a patch. Other studies [3, 10, 11]

automatically extracted likely program invariants based on dynamic program behavior.

Our work is different in the following ways: (1) we consider user environment and/or inputs as failure predictors; (2) we do not use static analysis to find bugs; rather, we use it to reduce the length of call sequences for environment-related bugs; (3) we use the commonality between execution profiles (or the lack thereof) as a failure predictor rather than the invariants over the executions; (4) we use the learned prediction model and execution similarity to prevent upgrade failures for future users; and (5) we restrict the execution of instrumented versions of the software to a very short time (just before or briefly after the upgrade).

5. Conclusion

We proposed Mojave, the first recommendation system for preventing software upgrade failures. Our evaluation with two OpenSSH upgrade failures shows that Mojave provides accurate recommendations to most users. Based on these positive initial results and the potentially high cost of failures, we conclude that Mojave can be useful in practice.

References

[1] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Practices and Techniques*. Addison-Wesley, 1986.

[2] BACHWANI, R., CRAMERI, O., BIANCHINI, R., KOSTIĆ, D., AND ZWAENEPOEL, W. Sahara: Guiding the Debugging of Failed Software Upgrades. In *Proceedings of International Conference on Software Maintenance* (2011).

[3] BRUN, Y., AND ERNST, M. D. Finding Latent Code Errors via Machine Learning over Program Executions. In *Proceedings of the International Conference on Software Engineering* (2004).

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

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

[6] CRAMERI, O., KNEZEVIC, N., BIANCHINI, R., KOSTIĆ, D., AND ZWAENEPOEL, W. Staged Deployment in Mirage, An Integrated Software Upgrade Testing and Distribution System. In *Proceedings of Symposium on Operating Systems Principles* (2007).

[7] DHALIWAL, T., KHOMH, F., AND ZOU, Y. Classifying Field Crash Reports for Fixing Bugs: A Case Study of Mozilla Firefox. In *Proceedings of the International Conference on Software Maintenance* (2006).

[8] DICKINSON, W., LEON, D., AND PODGURSKI, A. Finding Failures by Cluster Analysis of Execution Profiles. In *Proceedings of the International Conference on Software engineering* (2001).

[9] ENGLER, D., CHEN, D., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the International Symposium on Operating Systems Principles* (2001).

[10] ERNST, M., COCKRELL, J., GRISWOLD, W., AND NOTKIN, D. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of International conference on Software engineering* (1999).

[11] HANGAL, S., AND LAM, M. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of International Conference on Software Engineering* (2002).

[12] HILL, W., AND L. STEAD, M. ROSENSTEIN, G. F. Recommending and Evaluating Choices in a Virtual Community of Use. In *Proceedings of the Conference on Human factors in Computing Systems* (1995).

[13] KOHAVI, R. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the International Joint Conference on Artificial Intellligence* (1995).

[14] LANDWEHR, N., HALL, M., AND FRANK, E. Logistic model trees. In *Machine Learning* (2003).

[15] LIU, C., AND X. YAN, H. YU, J. H. P. Y. Mining Behavior Graphs for Backtrace of Noncrashing Bugs. In *Proceedings of the International Conference on Data Mining* (2005).

[16] MARKOV, Z., AND RUSSELL, I. An Introduction to the WEKA Data Mining System. In *Proceedings of Annual Conference on Innovation and Technology in Computer Science Education* (2006).

[17] MCCAMANT, S., AND ERNST, M. Predicting Problems Caused by Component Upgrades. In *Proceedings of European Software Engineering Conference* (2003).

[18] MCCAMANT, S., AND ERNST, M. Early Identification of Incompatibilities in Multi-component Upgrades. In *Proceedings of the European Conference on Object-Oriented Programming* (2004).

[19] MIRGORODSKIY, A., MARUYAMA, N., AND MILLER, B. Problem Diagnosis in Large-scale Computing Environments. In *Proceedings of the Conference on Supercomputing* (2006).

[20] NAGAPPAN, N., BALL, T., AND ZELLER, A. Mining Metrics to Predict Component Failures. In *Proceedings of the International Conference on Software engineering* (2006).

[21] NECULA, G., MCPEAK, S., RAHUL, S., AND WEIMER, W. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the International Conference on Compiler Construction* (2002).

[22] OBERHEIDE, J., COOKE, E., AND JAHANIAN, F. If It Ain't Broke, Don't Fix It: Challenges and New Directions for Inferring the Impact of Software Patches. In *Proceedings of the Hot Topics in Operating Systems* (2009).

[23] PODGURSKI, A., LEON, D., FRANCIS, P., MASRI, W., MINCH, M., SUN, J., AND WANG, B. Automated Support for Classifying Software Failure Reports. In *Proceedings of the International Conference on Software Engineering* (2003).

[24] QUINLAN, J. R. Induction of Decision Trees. *Machine Learning* (1986).

[25] RESNICK, P., IACOVOU, N., SUCHAK, M., BERGSTROM, P., AND RIEDL, J. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In *Proceedings of the Conference on Computer Supported Cooperative Work* (1994).

[26] SHARDANAND, U., AND PATTIE, M. Social Information Filtering: Algorithms for Automating "Word of Mouth". In *Proceedings of the Conference on Human Factors in Computing Systems* (1995).

[27] SU, Y., ATTARIYAN, M., AND FLINN, J. Autobash: Improving configuration management with operating system causality analysis. In *Proceedings of Symposium on Operating Systems Principles* (2007).

[28] TUCEK, J., LU, S., HUANG, C., AND S. XANTHOS, Y. Z. Triage: diagnosing production run failures at the user's site. In *Proceedings of Symposium on Operating Systems Principles* (2007).

[29] WANG, H., J. PLATT, Y. CHEN, R. Z., AND WANG, Y. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of the Symposium on Operating Systems Design and Implementation* (2004).

[30] YUAN, C., L. NI, J. W., LI, J., ZHANG, Z., WANG, Y., AND MA, W. Automated Known Problem Diagnosis with Event Traces. In *Proceedings of the European Conference on Computer Systems* (2006).