Sahara: Guiding the Debugging of Failed Software Upgrades

Rekha Bachwani, Olivier Crameri[†], Ricardo Bianchini, Dejan Kostić[†], and Willy Zwaenepoel[†]
Rutgers University
{rbachwan,ricardob}@cs.rutgers.edu
[†]EPFL
{olivier.crameri,dejan.kostic,willy.zwaenepoel}@epfl.ch

***Abstract*—Today, debugging failed software upgrades is a long and tedious activity, as developers may have to consider large sections of code to locate the bug. We argue that failed upgrade debugging can be simplified by exploiting the characteristics of upgrade problems to prioritize the set of routines to consider. In particular, previous work has shown that differences between the computing environment in the developer's and users' sites cause most upgrade problems. Based on this observation, we design and implement Sahara, a system that identifies the aspects of the environment that are most likely the culprits of the misbehavior, finds the subset of routines that relate to those aspects, and selects an even smaller subset of routines to debug first. To achieve its goals, Sahara leverages feedback from a large number of user sites, machine learning, and static and dynamic source analyses. We evaluate Sahara for three real upgrade problems with the OpenSSH suite, one synthetic problem with the SQLite database, and one synthetic problem with the uServer Web server. Our results show that the system produces accurate recommendations comprising only a small number of routines.**

I. Introduction

Modern software systems are complex and comprise many interacting and dependent components. Frequent upgrades are required for some or all components to fix bugs, patch security vulnerabilities, add or remove features, and other critical tasks. Unfortunately, many of the upgrades either fail or produce unwanted behavior. A survey conducted by Crameri *et al.* [9] showed that 90% of system administrators perform upgrades at least once a month, and that 5–10% of them is problematic. Interestingly, they also found that the most common source of upgrade problems 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. Such problems are difficult (or maybe impossible) to prevent because the developer cannot foresee, much less test her software for, every possible environment in which the software might be used.

When upgrades misbehave at some user sites, the developers receive bug reports and complaints. 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.

In this paper, we propose Sahara, a system that simplifies the debugging of environment-related upgrade problems by pinpointing the subset of routines and variables that is most likely the source of misbehavior. 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 targets C applications written for Unix-like operating systems.

In more detail, Sahara applies feature selection [35] 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. At this point, Sahara deploys instrumented versions of the current and upgraded version of the code 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 [36] 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. Our results demonstrate that Sahara produces recommendations that always include the routines responsible for the bugs. The exact number of recommended routines 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

This research was partially funded by NSF grant CSR-0720568 and Hasler Foundation grant 2103.

```
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. }
```

Fig. 1. Example.

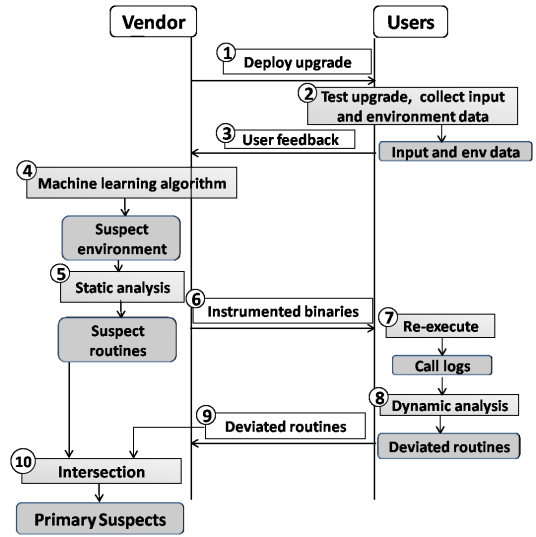

Fig. 2. Overview of Sahara.

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.

II. SAHARA: PRIORITIZING UPGRADE DEBUGGING

A. A Motivating Example

To make our exposition more concrete, let us look at a simple example in Fig 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 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 [28] that we detail in Section III-A.

The failure 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.

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 in behavior 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.

B. Design and Implementation

Overview. Figure 2 illustrates the steps involved in Sahara. First, Sahara deploys the upgrade to any users that request it (step 1). As the software executes at each user's site, Sahara collects information about the environment and inputs used (step 2). At the end of the execution, Sahara obscures and then transfers the collected environment information (the *inputs are never transferred* on the network) to the developer's site, along with a success/failure flag provided by the user (step 3). (Obviously, some users may decide not to allow any sort of information to be collected or provided to Sahara.) The information about the environment includes the version of the operating system, the version of the libraries, the configuration settings, the name and version of the other software packages installed, and a description of the hardware. A failure flag may mean that (a) the upgrade could not be properly installed or executed, (b) the upgrade caused incorrect behavior or a crash, or (c) the upgrade caused another software to misbehave [9].

Now suppose that the upgrade misbehaved at one user site at least. With the environment and success/failure information at the developer's site, Sahara runs a machine learning algorithm to determine the aspects of the environment that are most likely to have caused the misbehavior (step 4). Next, based on def-use static analysis, Sahara isolates the variables in the code that derive directly or indirectly from those aspects; the routines that use these variables are considered suspect (step 5).

Sahara then deploys instrumented versions of the current and upgraded code 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). Upgrade deployment in Sahara 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.

Sahara uses the Mirage tracing infrastructure, which has been detailed in [3], [9]. Thus, 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 following resources are considered as an application's environment: a) all files accessed read-only (such as configuration files) by the application; b) all files of a certain type (such as libraries); c) all files in the package being upgraded. Furthermore, Sahara provides an API that allows the developer to include or exclude files or directories. In addition to the data accessed during application execution, Sahara collects information about the hardware and software installed.

Again as in Mirage, Sahara provides parsers to compute a concise representation (fingerprint) for each environmental resource. The parsers know how to extract relevant information from a file based on its type and hash its content at a specific granularity. For instance, the parsers for binary files generate fingerprints at a coarser granularity than the parsers for a configuration file. We use SHA-1 to compute fingerprints of the resources. In each fingerprint, the name of the resource serves as a key and the hash of its contents as the value.

For the users who choose to participate, Sahara 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), Sahara collects the environment resource information and produces the fingerprints. After each of these executions, Sahara also queries the user about whether the upgrade has succeeded or failed. We ask for 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 [9].

When the user provides a succeed/fail flag, Sahara sends this information, along with the environment resource fingerprints, back to the developer's 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 III 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. The fingerprints are never "unhashed" during feature selection (or after it); it is enough for Sahara to know how many different fingerprints there are for each feature.

Sahara uses the decision tree algorithm with feature ranking from the WEKA tool [www.cs.waikato.ac.nz/ml/weka/] 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 [30], a measure of a feature's ability to create subsets with homogeneous classes. In Sahara, 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 Fig 1, the root feature 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.

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, Sahara uses 10-fold cross-validation [16] to compute the standard deviation of the ranks of each feature. When the standard deviations of the top-ranked features are high, Sahara warns the developer that its results are not to be trusted, i.e. the reason for the failures is unlikely to be the environment. When this condition is not met, 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 III.

Static analysis and suspect routines (step 5). Sahara analyzes the upgraded software using the *C Intermediate Language* (CIL) [24]. 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 shows the def-use chain (thin arrows) for our example program.

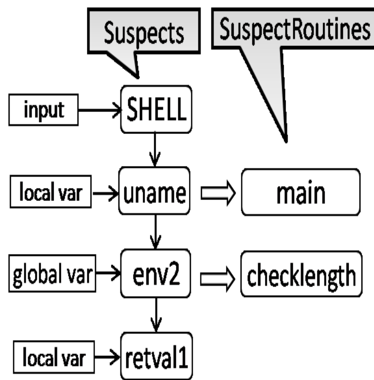

Fig. 3. Def-use chain, suspect variables and routines for our simple example.

To find suspect routines, Sahara traverses all the routines in the order they appear in the call graph, starting with the root. During the course of the traversal, Sahara maintains three lists: (1) a list of global suspect variables (*SuspectVars*); (2) a list of per-routine suspect variables (*LsuspectVars*); and (3) a list of routines that are suspect (*SuspectRoutines*). *SuspectVars* is initialized with the variables corresponding to SERs.

Sahara analyzes each routine statement-by-statement, starting with the root routine. For every variable access, it checks whether the variable is a suspect or depends on any suspect, either directly or indirectly. If so, the accessed variable becomes a suspect. If it is a local variable, it is added to *LsuspectVars* of the routine where the access appears; otherwise, it is added to *SuspectVars*. The routine containing the access is added to *SuspectRoutines*. In addition, if a routine calls another with a suspect variable as a parameter, the caller is added to *SuspectRoutines* and the corresponding formal parameter is added to the *LsuspectVars* of the callee. 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 *SuspectRoutines (SRs)*, after the entire graph has been traversed.

This step produces a set of routines that are highly correlated with the failures. For the example in Fig 1, `main` and `checklength` are the two suspect routines. The block arrows in Figure 3 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.

```
1.  Function main numArgs 0
2.  Globals at ENTRY: 0
3.      Function checklength numArgs 0
4.      Globals at ENTRY: 1
5.          Global: env2 Size: 4 Type: int Value: 9
6.      Globals at EXIT: 1
7.          Global: env2 Size: 4 Type: int Value: 9
8orig.      Return: retVal Size: 4 Type: int Value: 9
8upg.       Return: retVal Size: 4 Type: int Value: -1
9.
10.     Function secondfunction numArgs 1
11.     Globals at ENTRY: 1
12.         Global: glob Size: 4 Type: int Value: 3
13orig.     Param: a Size: 4 Type: float Value: 2.2
13upg.      Param: a Size: 4 Type: float Value: 5.1
14.     Globals at EXIT: 1
15.         Global: glob Size: 4 Type: int Value: 3
16.     Return: retVal Size: 4 Type: int Value: 10
17. Globals at EXIT: 0
18. Return: retVal Size: 4 Type: int Value: 0
```

Fig. 4. Execution log of two versions.

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 dynamic 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 [9]. 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. Fig 4 shows the log for the two versions. Since the logs of the two versions are mostly same (except for lines 8 and 15), only the lines that are different between the two versions are duplicated.

With these logs, Sahara determines the set of routines, called *DeviatedRoutines (DRs)*, whose behavior has deviated after the upgrade. Specifically, we implement *fDiff*, a tool that converts each of log into a sequence of routine signatures, and uses the longest common subsequence algorithm to compute the difference between the sequences. FDiff is similar to Unix's diff, but produces more concise output as it understands the call/return structure of our logs. A routine has deviated, if the following differs between the two versions: (1) its number of arguments; (2) the value of any of its arguments; (3) its return value; (4) the number of global variables it accesses; or (5) the value of one or more global variables it accesses. This notion of deviation is similar to that of value spectra [36]. Wilde and Scully [34] also compare execution logs.

In Fig 4, two routines have deviated: `checklength` has deviated in its return value (line 8), whereas `secondfunction` has deviated in its argument (line 13).

Sahara transfers the *DRs* list to the developer's site.

Intersection and list of primary suspects (step 10). Finally, Sahara computes the union of the *DRs* from the failed user sites. It then intersects this larger set with the *SRs*, thereby eliminating benign deviations that have nothing to do with the failure. 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`.

C. 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., [32]). 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 [9]).

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. [12].

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 *DRs* 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 limits the user information transferred to the developer's site to the resource fingerprints. In our current implementation, the fingerprints are transferred in hashed form (SHA-1), which does not provide foolproof privacy guarantees. However, Sahara can easily use more sophisticated schemes for these transfers. Regardless of the privacy scheme, the bandwidth required by these transfers (and that of the *DRs*) should be negligible. Sahara requires more bandwidth for transferring the re-execution and value spectra infrastructures, but only for failed user sites.

Sahara employs static and dynamic analyses to narrow the set of routines that are likely to contain the root cause of the failure. However, under certain conditions, these analyses may be unable to do so. In the worst case, all routines may be affected by the SERs, making static analysis ineffective. Similarly, 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 [39].

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., [17], [32], [39]) often incur substantial overhead at the users' sites, such as running instrumented code all the time, checkpointing state at regular intervals, and multiple replays.

III. Evaluation

In this section, we describe our methodology and evaluate Sahara by analyzing three real bugs in OpenSSH, a synthetic bug in SQLite, and a synthetic bug in uServer.

We chose OpenSSH because it is widely deployed in diverse user environments. Its upgrades are fairly frequent, typically once every 3–6 months [26]. 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).

SQLite is the most widely deployed SQL database [31]. It implements a serverless, transactional SQL engine. SQLite has 67K LOC spread across 4 files. uServer [7] is an open-source, event-driven Web server sometimes used for performance studies. It has 37K LOC spread across 161 files.

A. Methodology

OpenSSH: Port forwarding bug. Port forwarding is commonly used to create a SSH tunnel. To setup a 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 [5] 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's 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 [5].

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.

OpenSSH: X11 forwarding bug. This bug [4] manifested when users upgraded to OpenSSH version 4.2p1 from 4.1p1 and tried to start X11 forwarding. The following error was observed at the sites that had SSH forwarding support enabled and the command was executed in the background:

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

In version 4.2p1, developers modified the X11 forwarding code to fix some 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 [4].

OpenSSH: ProxyCommand bug. The ProxyCommand option specifies the command that will be used by the SSH client to connect to the remote server. The bug [28] was a regression in OpenSSH version 4.9; ssh with ProxyCommand would fail for some 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 [28].

SQLite and uServer bugs. To demonstrate Sahara'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* and could be identified by simpler tools than Sahara. However, *our goal is simply to demonstrate that Sahara works without modification for a variety of applications.*

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

We do not present complete results for the ProxyCommand, SQLite, or uServer bugs due to space limitations. However, we do include a summary of their results in the next subsection.

Upgrade deployment. To simulate a real-world deployment of a software upgrade to 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 differ across clusters.

We used the methodology described in Section II-B to identify the environmental resources in OpenSSH, SQLite, and uServer. Sahara uses the following parsers to parse and fingerprint the 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 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. The environmental resources of a single machine, parsed/chunked and fingerprinted, along with the success/failure flag constitute a single user profile.

By default, our experiments assume that 20 profiles include environment settings that can activate a bug, whereas 67 of them do not. We study the impact of this parameter below.

User site environments. To evaluate Sahara's behavior in the face of the uncertainties that may occur in practice, we perform six types of experiments: random perfect (*rand_p*), two random imperfect (*rand_i60* and *rand_i20*), real configuration perfect (*real_p*), and two real configuration imperfect (*real_i60* and *real_i20*). In the rand_p experiment, 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.

In the two rand_i cases, the environment settings are the same as in the rand_p 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 rand_i60 case, and only 20% in the rand_i20 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 rand_p case.

In the three types of experiments above, the application-related environment includes random values. For more realistic scenarios, we downloaded eight different complete OpenSSH configuration files from the Web. For each of the bugs, we modify three of these files to include the settings that activate the bug. One of these eight configuration files (three with problematic settings and five with only good settings) is assigned to each of the 87 user profiles randomly, but in the same proportion as before: 20 users should get problematic settings and 67 should not. In the real_p case, all the 20 profiles with problematic settings are labeled as failures, whereas the 67 others are labeled as successful. In the real_i60 and real_i20 experiments, only 60% and 20% of the profiles with these settings are labeled as failures, respectively. The real configurations are likely to lead to more SERs than the random ones. We do not study real configurations for SQLite and uServer because we inject synthetic bugs into them.

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

B. 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.

Bug	Experiment	diff	SERs	SRs	DRs	PSRs
Port	rand_p	65	1	12	124	6
	rand_i60	65	1	12	124	6
	rand_i20	65	1	12	124	6
	real_p	65	3	22	124	7
	real_i60	65	3	22	124	7
	real_i20	65	3	22	124	7
X11	rand_p	137	1	18	157	6
	rand_i60	137	1	18	157	6
	rand_i20	137	1	18	157	6
	real_p	137	3	21	157	7
	real_i60	137	3	20	157	6
	real_i20	137	3	20	157	6

TABLE I
OPENSSH BUG RESULTS. SERS = SUSPECT ENVIRONMENT RESOURCES; SRS = SUSPECTROUTINES; DRS = DEVIATEDROUTINES; PSRS = PRIME SUSPECT ROUTINES.

Table I 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 chosen as suspect in the rand_p, rand_i60, and rand_i20 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. Tunnel corresponds to 4 suspect variables in ssh.

In contrast, in the real_p, real_i60 and real_i20 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 real cases. The 12 routines comprise those that (1) read the configuration file and initialize the environment of the ssh client; (2) create, enable, or disable a tunnel; (3) place the tunnel data into a buffer or a packet; and (4) enable the port forwarding over this tunnel and create a channel for it. Routine *channel_new* from the latter group contains the root cause of this failure.

In the real cases, the same 12 routines are suspect, in addition to those affected by RSAAuthentication. 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 SRs and DRs is only 6 routines in the random cases and 7 routines in the real 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 real 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 I, 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 354 features.

The bottom-half of Table I presents the results. The feature selection step again results in 1 feature chosen as suspect in the rand_p, rand_i60, and rand_i20 cases. This feature is exactly the environment resource that is directly related to the bug: configuration parameter `X11Forwarding`. It corresponds to 3 variables in the sshd code.

In the real_p experiment, Sahara selects 3 features: configuration parameters `X11Forwarding`, `AuthorizedKeysFile`, and `ChallengeResponseAuthentication`. In the real_i60 and real_i20 cases, Sahara also selects 3 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 4 features correspond to 7 actual variables in sshd.

The static analysis results in 18 suspect routines in the rand_p and rand_i cases, 21 in real_p, and 20 in the real_i cases. The 18 routines comprise those that: (1) read the configuration file and initialize the environment of sshd; (2) authenticate the incoming client connection with the options specified and setup the connection; (3) start a packet for X11 forwarding; and (4) setup X11 forwarding, create the channel, process X11 requests, and do the cleanup. Routine *session_setup_x11fwd* from the latter group is the culprit.

In the real configuration cases, all the 18 routines mentioned above are suspect, in addition to those affected by AuthorizedKeysFile and ChallengeResponseAuthentication. 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 real_p case has 1 more routine reported as suspect than the two real_i 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 in the random case, and 7 in the real configuration 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.

From these results, we can see that the number of primary suspects found by Sahara is at least 3x lower than when using static analysis alone, at least 20x 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 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 II presents the "perfect" results from these experiments. The default results (rand_p and real_p) 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 real configuration results for the X11 forwarding bug show that lowering noise (going from real_p to real_10) can indeed improve results as well.

Impact of feature selection accuracy. Our longer technical report [3] also includes a study of the impact of feature selection accuracy on Sahara. In short, these results illustrate the behavior we expected: the less accurate feature selection

Bug	Experiment	SERs	SRs	DRs	PSRs
Port	rand_p_30	1	12	124	6
	rand_p	1	12	124	6
	rand_10	1	12	124	6
	real_30	1	12	124	6
	real_p	3	22	124	7
	real_10	3	22	124	7
X11	rand_p_30	1	18	157	6
	rand_p	1	18	157	6
	rand_10	1	18	157	6
	real_30	1	18	157	6
	real_p	3	21	157	7
	real_10	2	20	157	6

TABLE II
IMPACT OF #PROFILES WITH FAILURE-INDUCING SETTINGS. SERS = SUSPECT ENVIRONMENT RESOURCES; SRS = SUSPECTROUTINES; DRS = DEVIATEDROUTINES; PSRS = PRIME SUSPECT ROUTINES.

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 PSRs by 6x–7x.

OpenSSH: ProxyCommand bug. 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. We performed the same 10 experiments with this upgrade as above. Depending on the type of experiment, feature selection produces 2–5 SERs and static analysis produces 10–29 suspect routines. Dynamic analysis produces 284 deviated routines. In contrast, Sahara outputs 7 or 11 PSRs in all but one experiment (real_10, for which it recommends 21 routines). Overall, Sahara improves on static analysis by 1.4x and on dynamic analysis by 14x–40x for this bug.

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 *SRs*, and dynamic analysis identified 14 *DRs*. Sahara outputs 2 PSRs in each of the three random cases (exactly the routines that were modified); one of the PSRs 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 suggest that our system may significantly reduce the time and effort required to diagnose the root cause of upgrade failures.

IV. RELATED WORK

A. Upgrade Deployment and Testing

A few studies [9], [21], [22] have proposed automated upgrade deployment and testing techniques. McCamant and Ernst [21], [22] automatically identify incompatibilities when upgrading a component in a multi-component system. However, they did not attempt to isolate the root cause of the incompatibilities. Similarly, Crameri *et al.* [9] did not seek to determine the root cause of upgrade failures.

B. Automated Debugging

Troubleshooting misconfigurations. PeerPressure [33], Snitch [23], and ConfAid [2] seek to identify the root cause of software misconfigurations. These systems assume that the software is correct, but was misconfigured by users. Sahara is fundamentally different; it helps find upgrade bugs triggered by proper configurations and environments. Moreover, Sahara goes well beyond finding the environment resources most likely to be related to a bug (i.e., feature selection).

Qin *et al.* [29] 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 [32] goes further by dynamically changing the execution environment while attempting to diagnose failures at users' sites.

Sahara focuses on 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 II-C, Sahara is complementary to systems like Triage.

Statistical debugging with user site feedback. Several previous papers [8], [12], [18], [19], [20], [27], [39] rely on low-overhead, privacy-preserving instrumentation infrastructures to provide user execution data back to developers. These works do not consider the users' environment, and require users to constantly run instrumented code and send feedback back to the developers, both of which have overheads.

Sahara also relies on information gathered at user sites, but the data collection only lasts temporarily to lower overheads. In addition, Sahara restricts its statistical analysis (feature selection) to the aspects of the environment that may have caused an upgrade to misbehave. Finally, Sahara goes further by relating the results of the analysis to the variables and routines that most likely caused the misbehavior.

Delta debugging. Delta debugging aims to resolve regression faults automatically and effectively. Several studies [8], [15], [39] have 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.

Sahara's dynamic analysis also considers 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, Sahara requires neither instrumenting the production code nor replaying the execution multiple times at the users' sites.

Dynamic behavior deviations. Xie and Notkin [36] proposed program spectra to compare versions and get insights into their internal behavior. Harrold *et al.* [14] 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.

In [25], [38], the authors propose a search algorithm 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, e.g. [6], [10], [11], [13], [37]. Sahara is not closely related to any of these approaches, except peripherally for its use of static (def-use) or dynamic analysis. However, Sahara's use of static and dynamic analyses differs in a major way from most other approaches: it does not use them to find the bugs themselves; rather, it uses them to constrain the set of routines of interest.

V. CONCLUSION

In this paper, we sought to reduce the effort developers must spend to debug failed upgrades. We proposed Sahara, a system that prioritizes the set of routines to consider when debugging. Driven by the fact 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 our system for five bugs in three widely used applications. Our results showed that Sahara produces accurate recommendations with only a small set of routines. Importantly, the set of recommended routines remains small and accurate, even when the user site information is misleading or limited.

REFERENCES

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

[2] ATTARIYAN, M., AND FLINN, J. Automating Configuration Troubleshooting With Dynamic Information Flow Analysis. In *OSDI* (2010).

[3] BACHWANI, R., ET AL. Sahara: Guiding the Debugging of Failed Software Upgrades. Tech. Rep. DCS-TR-676, Dept. of Computer Science, Rutgers University, October 2010, Revised January 2011.

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

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

[6] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI* (2008).

[7] CHANDRA, A., AND MOSBERGER, D. Scalability of Linux Event-Dispatch Mechanisms. In *USENIX ATC* (2001).

[8] CLEVE, H., AND ZELLER, A. Locating Causes of Program Failures. In *ICSE* (2005).

[9] CRAMERI, O., ET AL.

[10] ENGLER, D., ET AL. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP* (2001).

[11] ERNST, M., ET AL. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *ICSE* (1999).

[12] GLERUM, K., ET AL. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *SOSP* (2009).

[13] HANGAL, S., AND LAM, M. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *ICSE* (2002).

[14] HARROLD, M. J., ET AL. An Empirical Investigation of the Relationship Between Spectra Differences and Regression Faults. *Journal of Software Testing, Verification and Reliability* (2000).

[15] JEFFREY, D., GUPTA, N., AND GUPTA, R. Fault Localization Using Value Replacement. In *ISSTA* (2008).

[16] KOHAVI, R. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *IJCAI* (1995).

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

[18] LIBLIT, B., ET AL. Bug Isolation via Remote Program Sampling. In *PLDI* (2003).

[19] LIBLIT, B., ET AL. Scalable Statistical Bug Isolation. In *PLDI* (2005).

[20] LIU, C., ET AL. SOBER: Statistical Model-based Bug Localization. *ESEC* (2005).

[21] MCCAMANT, S., AND ERNST, M. Predicting Problems Caused by Component Upgrades. In *ESEC* (2003).

[22] MCCAMANT, S., AND ERNST, M. Early Identification of Incompatibilities in Multi-component Upgrades. In *ECOOP* (2004).

[23] MICKENS, J., SZUMMER, M., AND NARAYANAN, D. Snitch: Interactive Decision Trees for Troubleshooting Misconfigurations. In *SysML* (2007).

[24] NECULA, G., ET AL. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC* (2002).

[25] NESS, B., AND NGO, V. Regression Containment through Source Change Isolation. In *COMPSAC* (1997).

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

[27] ORSO, A., ET AL. Gamma System: Continuous Evolution of Software After Deployment. In *ISSTA* (2002).

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

[29] QIN, F., ET AL. Rx: Treating Bugs as Allergies - A Safe Method to Survive Software Failures. In *SOSP* (2005).

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

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

[32] TUCEK, J., ET AL. Triage: Diagnosing Production Run Failures at the User's Site. In *SOSP* (2007).

[33] WANG, H. J., ET AL. Automatic Misconfiguration Troubleshooting with PeerPressure. In *OSDI* (2004).

[34] WILDE, N., AND SCULLY, M. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice 7*, 1 (1995).

[35] WITTEN, I. H., AND FRANK, E. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[36] XIE, T., AND NOTKIN, D. Checking Inside the Black Box: Regression Testing Based on Value Spectra Differences. In *ICSM* (2004).

[37] ZAMFIR, C., AND CANDEA, G. Execution Synthesis: A Technique for Automated Software Debugging. In *Eurosys* (2010).

[38] ZELLER, A. Yesterday, my program worked. today it does not. why? In *ESEC* (1999).

[39] ZELLER, A. Isolating Cause-Effect Chains from Computer Programs. In *FSE* (2002).