An Integrated Framework for Improving the Quality and Reliability of Software Upgrades

THÈSE N^{O} 5087 (2011)

PRÉSENTÉE LE 24 JUIN 2011

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES D'EXPLOITATION
PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Olivier CRAMERI

acceptée sur proposition du jury:

Prof. A. Wegmann, président du jury
Prof. W. Zwaenepoel, directeur de thèse
Prof. R. Bianchini, rapporteur
Dr C. Cadar, rapporteur
Prof. D. Kostic, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2011

Abstract

Despite major advances in the engineering of maintainable and robust software over the years, upgrading software remains a primitive and error-prone activity. In this dissertation, we argue that several problems with upgrading software are caused by a poor integration between upgrade deployment, testing, and problem reporting. To support this argument, we present a characterization of software upgrades resulting from a survey we conducted of 50 system administrators. Motivated by the survey results, we present Mirage, a distributed framework for integrating upgrade deployment, testing, and problem reporting into the overall upgrade development process.

Mirage's deployment subsystem allows the vendor to deploy its upgrades in stages over clusters of users sharing similar environments. Staged deployment incorporates testing of the upgrade on the users' machines. It is effective in allowing the vendor to detect problems early and limit the dissemination of buggy upgrades.

Oasis, the testing subsystem of Mirage, improves on current state-of-the-art concolic and symbolic engines by implementing a new heuristic to prioritize the exploration of new or affected code in the upgrade. Furthermore, interactive symbolic execution, a new approach exposing the problem of path exploration to the tester using a graphical user interface, can be used to develop new search heuristics or manually guide testing to important areas of the source code.

In spite of all of these efforts, some bugs are bound to remain in the software when it is deployed, and will be discovered and reported only later by the users. With the last component of Mirage, we consider the problem of instrumenting programs to reproduce bugs effectively, while keeping user data private. In particular, we develop static and dynamic analysis techniques to minimize the amount of instrumentation, and therefore the overhead incurred by the users, while considerably speeding up debugging.

By combining up-front testing, stage deployment, testing on user machines, and efficient reporting, Mirage successfully reduces the number of problems, minimizes the number of users affected, and shortens the time needed to fix remaining problems.

Keywords: Upgrade testing, Clustering of machines, Staged software upgrade deployment, Testing, Regression testing, Debugging, Bug Reporting, Symbolic Execution, Static Analysis

Résumé

Malgré d'importantes avancées dans l'ingénierie de logiciels robustes ces dernières années, mettre à jour un logiciel reste une activité primitive et sujette à de nombreux problèmes. Dans cette thèse, nous prétendons que de nombreux problèmes de mise à jour sont causés par une mauvaise intégration entre le déploiement, le test et le rapport de problèmes.

Pour supporter cet argument, nous présentons une caractérisation des mises à jour de logiciel obtenue à travers un sondage de cinquante administrateurs système. Sur la base des résultats du sondage, nous présentons Mirage, un système distribué intégrant le déploiement, le test et le rapport de problèmes dans le processus de développement de mises à jour.

Le sous-système de déploiement de Mirage permet au vendeur de déployer ses mises à jour par étapes réparties sur des groupes d'utilisateurs partageant des caractéristiques de leur environnement. Le déploiement par étapes incorpore le test des mises à jour sur les machines appartenant aux utilisateurs. C'est une méthode efficace pour détecter les problèmes rapidement et limiter la propagation de ces derniers à de nombreux utilisateurs.

Oasis, le sous-système de test de Mirage, améliore l'état de l'art dans les moteurs d'exécution symbolique en mettant en oeuvre une nouvelle heuristique d'exploration du nouveau code, ou du code affecté par la mise à jour. De plus, l'exécution symbolique interactive, une nouvelle technique qui expose le problème de l'exploration des chemins au testeur via une interface graphique, peut être utilisée pour développer de nouvelles heuristiques de recherche ou pour manuellement diriger le test vers des régions du code source importantes.

Malgré tous ces efforts, certains problèmes persisteront toujours dans les logiciels déployés, et seront découverts et rapportés plus tard par les utilisateurs. Avec le dernier composant de Mirage, nous étudions le problème de l'instrumentation de programmes pour permettre une reproduction facile des problèmes, sans compromettre la confidentialité des données. En particulier, nous développons de nouvelles méthodes d'analyse statique et dynamique permettant de minimiser la quantité d'instrumentation nécessaire, et par conséquent la surcharge imposée aux utilisateurs, tout en accélérant considérablement le déboguage de problèmes.

En combinant le test, le déploiement par étapes, le test sur les machines d'utilisateurs, et un système de rapport de problèmes efficace, Mirage réduit significativement le nombre de problèmes, l'exposition des utilisateurs à ces derniers, et raccourcit le temps nécessaire pour déboguer les problèmes restants.

Mots-clés: Test de mises à jour, Classification de machines, Déploiement de mises à jour par étapes, Test, Test de régression, Déboguage, Rapport de bogues, Execution symbolique, Analyse statique

Acknowledgments

This dissertation is the culmination of an exciting albeit long and difficult journey. There are many people that have been a part of my life during those years. They all played an important role in allowing me to get to the end.

First and foremost, I would like to thank Aline, my girlfriend. She made tremendous efforts and sometimes even put aside her own busy life so that I could focus on my work. Her tireless support and kindness provided me with the necessary moral relief to keep moving forward.

I would like to thank my mom and dad for their unconditional support and unwavering belief that I would be successful in my life. I am extremely lucky to have them.

Getting through a Ph.D. requires a significant amount of work. I could always count on the help of my advisor, Willy Zwaenepoel, despite his already busy schedule as a Dean. Aside from Willy, I was very fortunate to work with Ricardo Bianchini, whom I now consider my mentor and friend. Both Willy and Ricardo were instrumental in helping me find a topic that was interesting and relevant. They invested significant efforts in my work and never let me down, even when it involved working at unexpected hours in various parts of the world to make a deadline. Thank you Willy and Ricardo!

I had the pleasure to collaborate with talented students and professors during my Ph.D. I would like to thank Nikola Knezevic, Rekha Bachwani, Dejan Kostic and Tim Brecht for working with me.

There is more to life than just work. Many thanks to Gilles Dubochet, Nicolas Jones and Renault John Lecoultre for providing me with an endless supply of coffee breaks at EPFL during which we always had fun and stimulating conversations. I would also like to thank all my friends outside of EPFL for bearing with me and my deadlines.

Finally, I thank my past and current colleagues in the lab: Aravind Menon, Steven Dropsho, Sameh Elnikety, Rodrigo Schmidt, Ming Iu , Emmanuel Cecchet, Denisa Ghita, Katerina Argyraki, Simon Schubert, Dan Dumitriu, Jiaqing Du, and Mihai Dobrescu. They all contributed in one way or another to my work, and I had a really great time interacting with them over the years.

Contents

Chapter 1

Introduction

Software upgrades involve acquiring a new version of an application, an improved software module or component, or simply a "patch" to fix a bug, and integrating it into the local system. Upgrades for Linux and Windows are released almost every month, while upgrades of user-level utilities and applications are frequent as well.

For software users, system administrators, and operators (hereafter collectively called users, for conciseness), integrating upgrades is often an involved proposition. First, an upgrade can affect multiple applications, and all of them may need to be tested for correct behavior after the upgrade. Second, the effects of buggy upgrades may need to be rolled back. Third, upgrades are prone to a variety of incompatibility and unexpected behavior problems, especially when components or modules, rather than entire applications or systems, are upgraded.

For software vendors, distributors, and open-source contributors (hereafter collectively called vendors, for conciseness), it is difficult to deploy the upgrades with high confidence that they will integrate properly into the users' systems and that they will behave as users expect. Recent significant advances in testing, such as symbolic execution, can improve the quality of the software upgrade. They suffer however from severe scalability problems and are not tailored to regression testing of software upgrades. For these reasons, vendors simply cannot anticipate and test their upgrades for all the applications and configurations that may be affected by the upgrades at the users' machines. To lessen this problem, vendors sometimes rely on "beta testing". However, beta testers seldom provide complete coverage of the environments and uses to which upgrades will be exposed. Another approach to reduce integration problems is for vendors to use package-management systems to deploy their upgrades. Dependency enforcement in these systems, however, only

tries to enforce that the right packages are in place. They do not provide any help with locally testing the upgrades for correct behavior or reporting problems back to vendors.

When problems occur, vendors receive only limited and often unstructured information to pinpoint and correct the problems. Most of the time, the vendor at best receives core dumps of applications that crash because of the upgrade or often incomplete problem reports posted to mailing lists or Web forums. Because of this limited information, it may take several iterations of deployment, testing, and debugging before an upgrade becomes useful to all of its intended users. Furthermore, because reports may come from any source, e.g., multiple beta testers or regular users with similar installations, there is often much repetition in the information received by the vendor, requiring significant human resources to filter out.

Although much anecdotal evidence suggests a high frequency of upgrade problems, there is surprisingly little information in the literature characterizing upgrades in detail. To start bridging this gap in the literature, we perform a characterization of upgrades, using responses to a survey that we conduct among 50 system administrators. The results confirm that upgrades are done frequently, that problems are quite common, that these problems can cause severe disruption and that therefore upgrades are often delayed, and that users seldom fully report the problems to the vendor. Additionally, many of the problems are caused by differences between the environment at the vendor and at the users. Broken dependencies, incompatibilities with legacy applications, and improper packaging issues are among this class of problems.

In this dissertation, we argue that the problems plaguing software upgrades are fundamental. To improve the quality of upgrades, and reduce the impact of problems, it is therefore necessary to revise the entire software upgrade life cycle. To this end, we present Mirage, a framework that integrates upgrade deployment, testing, and problem reporting. Figure 1.1 shows a high level overview of the components of Mirage. In the current state of the art, these three activities are loosely linked in an ad-hoc manner. Mirage integrates them into a structured and efficient upgrade development cycle that encompasses both vendors and users. Mirage tests upgrades before deployment using state of the art concolic execution. Then, it deploys complex upgrades in stages based on the clustering of the user machines according to their environments. It tests the upgrades again at the users' machines, and sends information back to vendors regarding the success/failure of each user's upgrade.

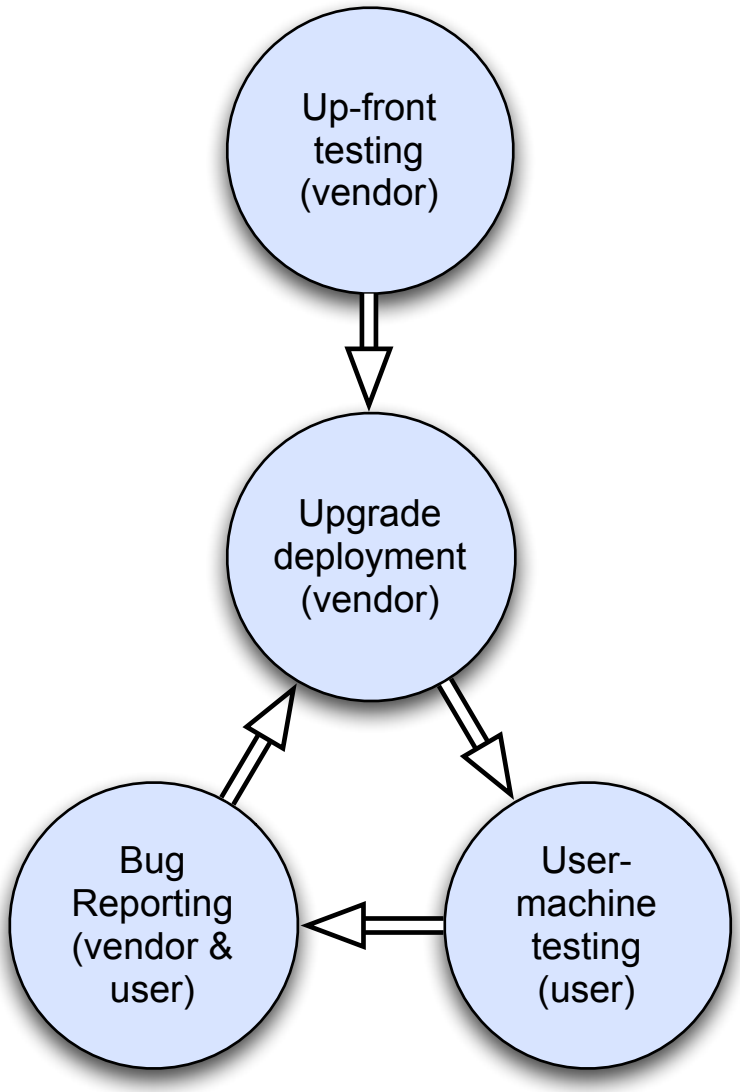

Figure 1.1: High-level overview of Mirage

In case of failure, whether during staged deployment or later, Mirage sends back a trace of branches simplifying reproduction of the problem. This trace is obtained using lightweight instrumentation to minimize the instrumentation overhead and only collect information that is determinant in helping the vendor reproducing the problem. By combining up-front testing, staged deployment, testing on user machines, and efficient reporting, Mirage successfully reduces the number of problems, minimizes the number of users affected, and shortens the time needed to fix remaining problems.

A key goal of Mirage is to guarantee that upgrades behave properly before they are widely deployed. Clustering machines as in Mirage helps achieve this goal, while simplifying debugging at the vendor. We also identify a fundamental trade-off between the number of upgrade problems at the user machines and the speed of deployment. To address this trade-off, Mirage provides a few abstractions on top of which different staged deployment protocols can be implemented.

In addition to staged deployment, Mirage provides a testing facility and a bug reporting system. Both components make extensive use of symbolic execution as well as dynamic and static analysis. We have designed and implemented Oasis, a state-of-the-art concolic execution engine (a variation of symbolic execution) specifically built to support user-machine testing and bug reporting in Mirage.

For testing, Oasis improves on state-of-the-art symbolic execution by implementing techniques to focus the testing on areas of the code affected by the changes in the upgrade. Our first technique is a new search heuristic that identifies the changes in the upgrade. It then uses a control-flow graph to direct the testing of the code to areas that have been modified. During each execution of the program, the heuristic tracks the effect of the changed code on the old code and subsequently strives to test the affected code more intensively. Our second technique is interactive symbolic execution, a new approach that exposes the problem of path explosion to the tester through a graphical user interface. This interface allows the tester to better understand the bottlenecks in trying to cover interesting parts of the code, and is therefore very useful in developing new search heuristics. Interactive symbolic execution can also be used to manually influence the exploration of the code by interfering with the scheduling of new paths to be explored.

Finally, to improve over the ad-hoc reporting seen in the survey, Mirage provides a structured reporting facility, which can use different combinations of static analysis and symbolic execution to

balance the trade-off between instrumentation overhead and debugging time. It allows users to report problems by sending the vendor a partial branch trace. This trace allows the vendor to easily reproduce the problem, therefore significantly speeding up debugging.

Overview of the results

We evaluate Mirage's clustering algorithm with respect to real upgrades and problems reported in our survey. The results demonstrate that our algorithm can cluster the users' machines effectively. Our algorithm works best when used in combination with vendor-provided information about the importance of certain aspects of the application's environment. Further, our simulations show that deploying upgrades in stages significantly reduces the amount of testing effort required of the users, while quickly deploying the upgrade at a large fraction of machines. Although we occasionally delay deploying certain upgrades, by increasing the users' confidence, they may actually apply the upgrades sooner.

The testing component of Mirage, Oasis, is evaluated using an open source web server. We show that our heuristic targeting new or affected code is able to test the upgrade much more intensively without sacrificing code coverage. Our preliminary experiments with interactive symbolic execution allowed us to better understand the progression of the heuristic and identify key areas where improvements are needed.

For bug reporting, we study the trade-off between instrumentation overhead and debugging time using an open-source Web server, the *diff* utility, and four coreutils programs. Our results shows that using a combination of dynamic and static analysis, our system is capable of limiting both the overhead of branch logging and the bug reproduction time. We conclude that our techniques represent an important step in improving bug reporting and making symbolic execution more practical for bug reproduction.

Contributions

At a high level, the main contributions of this thesis are the following:

1. The characterization of software upgrade problems using a survey of fifty system administrators.

2. The design, implementation and evaluation of Mirage, a framework that integrates deployment, user machine testing and

bug reporting into a structured and efficient upgrade development cycle that encompasses both vendors and users.

3. The design, implementation and evaluation of Oasis, a state-of-the-art concolic execution engine. Oasis supports regression testing of software upgrades through a directed search heuristic and a new technique called interactive symbolic execution.

4. The study of the trade-off between instrumentation overhead and debugging time for bug reporting using different techniques including symbolic execution, dynamic and static analysis.

We detail each of the contributions in the corresponding chapters of the dissertation.

Organisation of the dissertation

The rest of this dissertation is organized as follows. Chapter 2 presents the results of a survey characterizing upgrade problems. Chapter 3 describes our technique for staging upgrade deployment in order to reduce the impact of problems. Chapter 4 describes our implementation of Oasis, a state-of-the-art concolic engine and its application to regression testing. Chapter 5 describes our bug reporting technique using static and dynamic analysis to balance the trade-off between debugging time and instrumentation overhead. Chapter 6 presents the related works. Finally, Chapter 7 concludes the dissertation.

Chapter 2

Characterizing Upgrades

2.1 Methodology

We conducted an online survey to estimate the frequency of software upgrades, the reasons for the upgrades, the frequency of software upgrade problems, the classes of problems that occur, and the frequency of the different classes. We posted the survey (reproduced in appendix A) to the USENIX SAGE mailing list, the Swiss network operators group, and the EliteSecurity forum in Serbia. We received 50 responses.

The majority of the respondents to our survey (82%) have more than five years of administration experience, and 78% manage more than 20 machines. Our respondents were allowed to specify multiple, non-disjoint choices of operating systems, making it difficult to correlate upgrade problems with an operating system version. In our survey, 48 of the administrators manage a Linux system or some other UNIX-like system, 29 administrators manage Windows desktops and server platforms, while 12 respondents manage Mac OS machines.

2.2 Survey Results

Frequency of Software Upgrades

As Figure 2.1 demonstrates, upgrades are frequent. Specifically, 90% of administrators perform upgrades once a month or more often.

Reasons for Upgrades

We asked respondents to rank (from 1 to 5, 1 being most important) each of five possible reasons: 1) bug fix, 2) security patch,

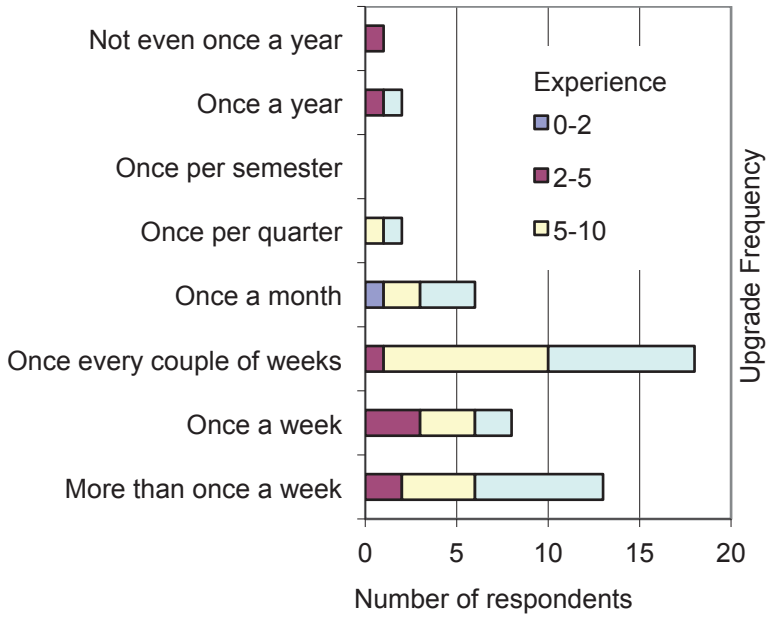

Figure 2.1: Upgrade frequencies.

3) new feature, 4) user request, and 5) other reason. Security fixes get the highest priority (average rank 1.6), followed by bug fixes (average rank 2.2) with user requests and new features trailing (average ranks of 3.3 and 3.5, respectively). Given that the primary reason for performing an upgrade is a security fix, it can be harmful if administrators delay upgrading.

Upgrade Installation Delays

Nevertheless, 70% of our respondents report that they refrain from installing a software upgrade, regardless of their experience level. This is the case even though 70% of administrators have an upgrade testing strategy (Figure 2.2 shows these results). An overwhelming majority (86%) of administrators use the software packaged with the operating system to install upgrades.

Upgrade Failure Frequency

We asked respondents to estimate this rate. As Figure 2.3 shows, 66% of them estimate that between 5 and 10% of the upgrades that are applied have problems. Given the potential damage and extra effort that a faulty upgrade can entail (discussed next), such a failure rate is unacceptable. Over all the responses, the average failure rate is 8.6% and the median is 5%.

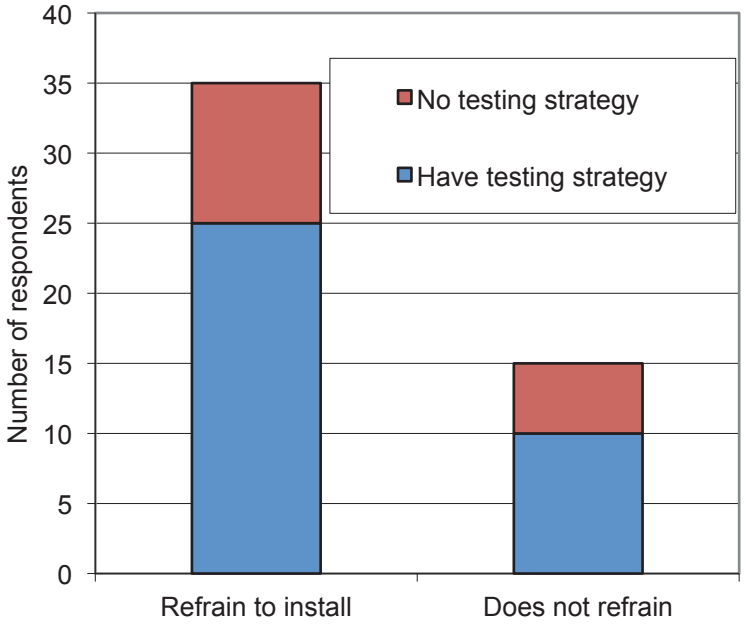

Figure 2.2: Reluctance to upgrade.

Another reason for avoiding upgrades may be the damage that a faulty upgrade can cause. An important fraction of the administrators (48%) experience problems that pass initial testing, with 18% reporting catastrophic upgrade failures.

Testing Strategies

The majority of the testing strategies our respondents use involve having a testing environment (25 respondents). Six of the administrators report testing the upgrade on a few machines, then moving on to a larger fraction, before finally applying the upgrade to the entire machine set. However, only 4 respondents use an identical machine configuration in the testing environment. Two of the respondents rely on reports of successful upgrades on the Internet without an additional testing strategy. Although 70% of the respondents have an upgrade testing strategy, only 18% of the administrators report that they do not have upgrade problems because they use it.

Causes of Failed Upgrades

We asked the administrators to assign a rank to each of the failure categories that we provided, including an optional "other categories" field. Our respondents identified broken dependencies, removed be-

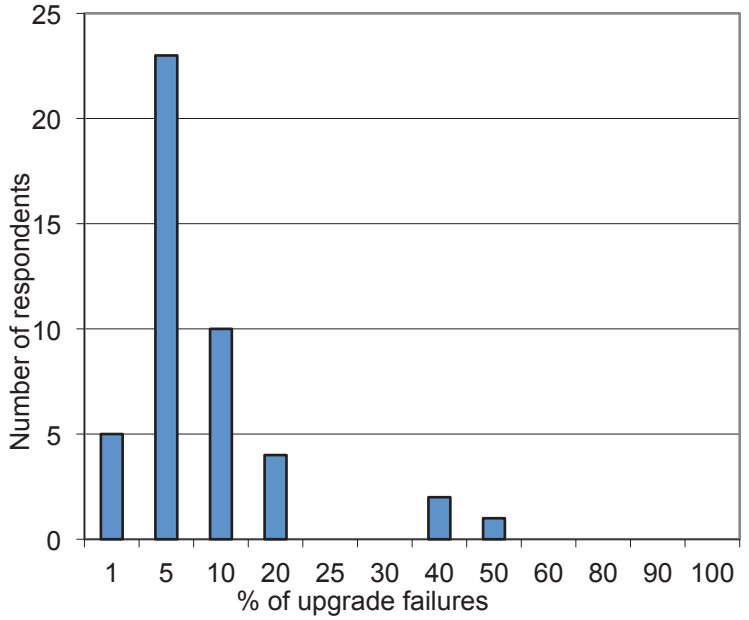

Figure 2.3: Perceived upgrade failure rate.

havior, and buggy upgrades as the most prevalent causes (average ranks of 2.5, 2.5 and 2.6, respectively). They deemed incompatibility with legacy configurations and improper packaging to be less common (average ranks of 3.1 and 3.2, respectively). We detail some of the reported problems in Section 2.3. Here we conclude that no single cause of upgrade failure is dominant.

Problem Reporting

Only half of the respondents consistently report upgrade problems to the vendor. When they do so, the information they typically include is the versions of the operating system and related software, type of hardware, and any error messages.

2.3 Categories of Upgrade Problems

Here we describe the major categories of upgrade problems (in decreasing order of importance assigned by respondents) and a few examples. We combined the reported problems with some that we found on Web forums.

Broken Dependency

This category captures subtle dependency issues that often cause severe disruption. Typically, application A_X is compiled with a specific library L_Y or depends on some application A_Y. Upgrading some other application A_Z that depends on L_Y (or A_Y) can upgrade L_Y (or A_Y) as well, causing application A_X to fail. For example, when upgrading the MySQL DBMS from version 4 to 5, the PHP scripting language crashes if it was compiled with MySQL support [41]. PHP still tries to use libraries from the old version. Some of these broken dependency issues could be caught by the package management system. However, package-management systems cannot catch these issues if the administrator upgrades some files manually.

Important Feature was Removed, or Behavior was Altered

A significant number of upgrade problems is caused by the removal of an important feature of the software. A related problem is altered behavior of an application or a changed API. An example of this category is an upgrade of PHP4 to PHP5 that caused some scripts (.php pages) to stop working due to a new Object Model [48]. The problems in this category cannot be alleviated by improved packaging, as it becomes almost impossible to search and upgrade all cases of deprecated behavior across all user files.

Buggy Upgrades

A natural category of the failed upgrades captures the bugs in the upgrade itself. For example, Firefox crashes when displaying some Web pages with JavaScript on Ubuntu, after the 1.5.0.9 upgrade is applied [21]. Some of the problematic upgrades our survey respondents reported have serious consequences, such as wireless, RAID, and other driver upgrades that caused the system to fail to boot.

Incompatibility with Legacy Configurations

Another category of upgrades involves machine-specific data that causes an upgraded application to fail. This kind of data includes environment variables and configuration files. For example, the upgrade of Apache from 1.3.24 to 1.3.26 caused Apache configurations that included an access control list to fail [2]. The contents of the included file had to be moved to the main configuration file for Apache

to start working again. Again it is difficult for upgrade packaging to handle all possible configurations at user machines.

Improper Packaging

This category of upgrade problems occurs when the creator of the new package (that contains the upgrade) does not correctly upgrade all application components. Typically, this includes an omission of some application file, as it was the case with SlimServer 6.5.1. Here, the database was not automatically upgraded during the upgrade process, and the server consequently would not start due to a changed database format. Issues like this one can be addressed by a more careful packaging procedure.

Minor Problems

The final category comprises problems that are not obvious bugs but represent an annoyance. Often, the problem arises when a cached file is overwritten instead of upgraded. Some of these seemingly minor problems can have serious consequences. For example, one administrator was unable to log into the Zertificon application due to an administrator password that was overwritten.

2.4 Discussion

We did not seek to perform a comprehensive, statistically rigorous survey of upgrade management in the field. Our main goals were to motivate Mirage, while collecting a sampling of data on real upgrades and their problems. These restricted goals allowed to focus on a group of administrators, mostly from USENIX SAGE, that volunteered information about their practical experiences. Admittedly, this approach may have been affected by self-selection, bias, and coverage problems. Nevertheless, our survey does provide plenty of information on real upgrades and is broader than the few comparable works in the literature [5, 50], which focused solely on security upgrades and have their own limitations. In fact, these other works confirm some of our observations: Beattie et al. [5] state that the initial security upgrade failure rate is even higher than that reported by our respondents for all upgrades. An inspection of 350,000 machines by Secunia [50] finds 28% of all major applications to be lacking the latest security upgrades, which is in line with the high fraction of our respondents who delay upgrades.

Chapter 3

Staged Deployment and Clustering

3.1 Introduction

Many approaches are available to reduce the burden of upgrades. For instance, improved development and testing methods at the vendor may reduce the number of buggy upgrades. As can be seen, however, from the results of the survey, several of the frequently occurring categories of upgrade problems are due to differences between the development and testing environment at the vendor and the many different environments at user machines. Broken dependencies and incompatibility with legacy configurations clearly fall into this category, whereas improper packaging may also result from environmental differences.

The goal of the Mirage project is to reduce the upgrade problems associated with such environmental differences between vendors and users. To do so, the Mirage framework provides three components that cooperate in a structured manner: staged deployment, testing, and problem reporting (see Figure 1.1). This chapter describes in details the staged deployment and clustering component of Mirage. It assumes that users are able to test upgrades before installing them and report success or failure. The next chapter describes a system capable of fulfilling this task automatically.

To correct problems with environmental differences between vendor and users, an upgrade would need to be tested in all possible user environments. In-house vendor testing of all possible user environments is infeasible, because the vendor cannot possibly anticipate all possible uses and environments. Exhaustive testing techniques such as symbolic execution may help, however they currently do not scale to real life software, and are unlikely to in the foreseeable

future. Indiscriminate testing at all user sites is also undesirable, because the problematic upgrades may inconvenience a large number of users. The current practice of beta-testers tries to reduce this inconvenience, but provides only limited coverage.

To address these issues, Mirage provides staged deployment. Machines are clustered based on their environments such that machines in the same cluster are likely to behave the same with respect to an upgrade. Within clusters, one or a few machines (called representatives) test the upgrades first, before other machines are allowed to download and test the upgrade. Finally, staged deployment allows control over the order in which upgrades are deployed to clusters.

With staged deployment, the vendor does not need to anticipate all possible user environments, but at the same time indiscriminate user-machine testing is avoided. With clustering, the representatives can provide better coverage than current beta-testing. As the survey shows, users are typically willing to wait to install upgrades, so the extra delay potentially introduced by staging is acceptable, especially in light of the reduction in inconvenience.

The rest of this chapter is organized as follows. Section 3.2 presents the design and implementation of our clustering and staged deployment techniques. Section 3.3 discusses the current limitations. Section 3.4 evaluates our clustering algorithm in the context of realistic upgrade problems, and analyzes the performance of two staged deployment protocols using a simulator. Finally, section 3.5 concludes the paper.

3.2 Design and Implementation

Mirage provides a small number of abstractions to support staged deployment. Upon these basic abstractions, vendors can build different deployment protocols to reach different objectives. Objectives may include reducing upgrade overhead (which we define as the number of machines that test a faulty upgrade), reducing upgrade latency (the delay between the upgrade being available at the vendor and it being applied at the user), reducing the number of redundant problem reports, front-loading the problem debugging effort, or combinations thereof.

Abstractions

The three basic deployment abstractions provided by Mirage are: clusters of deployment, representatives, and distance between ven-

dor and clusters. (1) User machines are clustered into groups that are likely to behave similarly with respect to upgrades of a particular application. (2) Each cluster has at least one representative machine. The representative tests the upgrade before any of the non-representative machines in its cluster do, playing a role similar to today's beta-testers. (3) In addition, a measure of distance may be defined between the vendor and a cluster, indicating the degree of difference between the vendor's environment and that of the machines in the cluster. Intuitively, if a machine is more dissimilar from the vendor's machine, the likelihood of a problem with the upgrade is higher. The vendor can take this additional information into account in guiding the deployment protocol. These three abstractions, and in particular the underlying clustering algorithm, are the main focus of this chapter.

Clustering is done per application, but has to be evaluated in the context of a particular upgrade. The quality of the clustering is critical for many of the potential vendor objectives. For example, the upgrade overhead is directly related to the quality of the clustering. To see this relationship, consider the following types of clustering assuming that user-machine testing detects an upgrade problem if and only if there is one.

In an *ideal* clustering, all machines in a cluster behave identically with respect to an upgrade and differently from machines in other clusters. If the upgrade behaves correctly at one machine in a cluster, it behaves correctly at all machines in that cluster. Further, all machines at which the upgrade behave correctly are in the same cluster. If the upgrade exhibits a problem, it exhibits that same problem everywhere in the cluster. Moreover, all machines that exhibit that problem are in the same cluster. If other machines exhibit a different problem, they are in a different cluster. Thus, upgrade overhead is limited to the number of representatives, which is minimal, as is the number of reports sent to the upgrade result repository.

While ideal, this form of clustering is difficult to achieve in practice. For this reason, we consider the following slightly less ambitious goal. In a *sound* clustering, all machines in a cluster behave identically with respect to an upgrade, as with ideal clustering, but there may be multiple clusters with the same behavior. In other words, multiple clusters may exhibit correct behavior or the same incorrect behavior. The upgrade overhead is no longer minimal, but as long as the the number of clusters with problems remains small relative to the total number of machines, there is considerable

improvement.

Clustering becomes considerably worse, if machines within the same cluster behave differently with respect to an upgrade. In this *imperfect* clustering, if an upgrade problem does not manifest itself at the representative, non-representa-tives are bothered with testing incorrect upgrades. This problem can be marginally improved by having a few rather than one representative. If the upgrade fails at the representative but works correctly at some other machines, then upgrades to those machines are needlessly delayed.

Protocols

Using these basic abstractions and a particular clustering algorithm, the vendor may implement different deployment protocols to optimize different criteria. In terms of structure, a protocol is defined by the degree of deployment parallelism it embodies and, when certain phases are sequential, the particular sequential orders it imposes.

In any staged deployment protocol, the upgrade is tested on the representative(s) of a cluster before it goes to non-representatives of that cluster. That constraint leaves open many variations in terms of parallelism and ordering.

A vendor may wait for all representatives of all clusters to successfully test an upgrade before it sends it to any non-representatives. Within this basic paradigm, it can send the upgrade in parallel to all or some representatives or sequentially one at the time to each of them. The same applies applies to the non-representatives: the upgrade may be sent to all or some clusters in parallel, or sequentially to one at a time.

An alternative approach is for the vendor to send the upgrade to the non-representatives of a cluster as soon as testing finishes successfully on the representatives of that cluster. This approach again leaves open the choice of sending the upgrades to the representatives of a cluster sequentially or in parallel.

When deployment should proceed one cluster after the other, the protocol needs to define the exact ordering of cluster deployments. To support this ordering, the vendor can use Mirage's distance metric. Going from smaller to larger distances may reduce the average upgrade latency, whereas going in the inverse order may allow the vendor to front-load its problem debugging effort.

Different upgrades may be treated differently. In fact, the vendor's objective for each upgrade typically depends on the characteristics of the upgrade. If the upgrade is a major new release, the vendor may decide to go slowly. If the upgrade is urgent and the

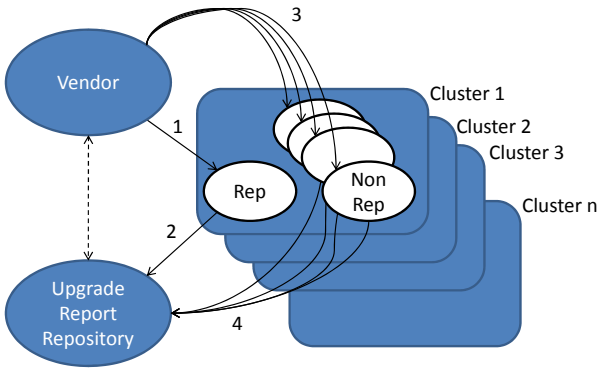

Figure 3.1: Mirage deployment: clusters of machines, cluster representatives and non-representatives, and cluster ordering. Some interactions of a deployment protocol are also shown.

vendor has high confidence in its correctness, it may bypass the entire cluster infrastructure, and distribute the upgrade to all users at once. Figure 3.1 depicts the key Mirage deployment abstractions and a few interactions of a generic deployment protocol.

Mirage does not advocate any specific protocol. In Section 3.4 we evaluate two specific protocols to illustrate some of the possibilities, but many other variations exist.

Clustering Machines

For upgrade deployment, Mirage seeks to cluster together user machines that are likely to behave similarly with respect to an upgrade. Since many of the most common upgrade problems result from differences in environment at the vendor and at the user machine, Mirage clusters machines with similar environments. Thus, before clustering, Mirage needs to identify the *environmental resources* on which the application to be upgraded depends. These resources typically include the operating system, various runtime libraries, executables, environment variables, and configuration files that each application uses (the environmental resources on a Windows-based system would include the registry as well).

Next, we detail the steps taken to collect and fingerprint (derive a compact representation) the set of environmental resources. After that, we describe our clustering algorithm.

Identifying Environmental Resources

We instrument, both at the vendor and at the user machines, the process creation, read, write, file descriptor-related and socket-related system calls. For environment variables, we intercept calls to the

`getenv()` function in `libc`. Using this instrumentation, we create a log of all external resources accessed. This log may include data files in addition to environmental resources. Clustering becomes ineffective if data files are considered as environmental resources.

We use a four-part heuristic to identify environmental resources among the files accessed by the application in a collection of traces, combined with an API that allows the vendor to explicitly include or exclude sets of files. The heuristic identifies as environmental resources: (1) all files accessed in the longest common prefix of the sequence of files accessed in the traces, (2) all files accessed read-only in all execution traces, (3) all files of certain types (such as libraries) accessed in any single trace, and (4) all files named in the package of the application to be upgraded. In our current implementation, we concentrate on files and environment variables. We plan to address machine hardware properties, network protocols, and other aspects of the environment in our future work.

The first part of the heuristic is based on the observation that the execution of applications typically starts with a single-threaded initialization phase in which the application loads libraries, reads configuration files and environment variables, etc. Most often, no data files are read in this initialization phase, and hence all files accessed in this phase are considered environmental resources. To find the end of the initialization phase, we compute the longest common sequence of files from the beginning that are accessed in every trace.

The second and third parts of the heuristic deal with applications that access environmental resources after the initialization phase, for example, applications that use late binding or load extensions at runtime. To identify those environmental resources, our heuristic considers all files opened read-only after the initialization sequence. To separate environmental resources from read-only data files, we only classify as environmental resources those files that are opened in every execution or those files that are of a few vendor-specified types, such as libraries.

The final part of our heuristic includes all files bundled in the package of the application to be upgraded.

The heuristic may erroneously designate certain files as environmental resources. For instance, the initialization phase may access a log or some initial data files, or sample data files may also be included in a package. Conversely, certain environmental resources are accessed in read-write mode and therefore not included by the heuristics, typically because the files contain both (mutable) data

and configuration information. To address these issues, a simple API provided by Mirage allows the vendor to include or exclude files or directories. By default, we exclude some system-wide directories, such as `/tmp` and `/var`.

As we show in Section 3.4, our heuristic combined with a very small number of vendor-provided API directives allows Mirage to correctly identify the environmental resources of popular applications with no false positives and no false negatives.

Resource Fingerprinting

To produce inputs to the clustering algorithm, we need a concise representation (a fingerprint) for each environmental resource at the vendor and at the user machines. Depending on its type, we have three different ways of fingerprinting a resource. First, for common types such as libraries, Mirage provides a parser that produces the fingerprint. Second, the vendor may provide parsers for certain application-specific resources, like configuration files. Third, if neither Mirage nor the vendor provides a parser for a resource, the fingerprint is a sequence of hashes of chunks of the file that are content-delineated using Rabin fingerprinting. In general, we expect that for many applications the vast majority of the resources are handled by parsers, and we resort to Rabin fingerprinting only in rare circumstances.

A resource's fingerprint contains a hierarchical set of keys and their values (items for short). The items produced by the parsers have a common structure but a meaning that is dependent on the resource type. Each parser is responsible for determining the granularity of the items that are produced. Indeed, producing fine-grained items might be useful for some types of resources (such as configuration files) and useless for others (such as executables). In addition, some resources may contain user-specific data (such as user names and IP addresses) and other information (such as comments) that are irrelevant in clustering machines; it is up to each parser to extract the useful information from the resource and discard the rest.

The parsers for the most common resource types produce items in the following formats:

- Executables: Executablename.FILE_HASH
- Shared libraries: LibraryName.Version#.HASH
- Text files: Filename.Line#.LINE_HASH
- Config files: Filename.SectionName.KEY.HASH
- Binary files: Filename.CHUNK_HASH

In contrast, the content-based fingerprinting creates items of the form: Filename.CHUNK_HASH. Clearly, these fingerprints are more coarsely grained than what is possible with parsers. In fact, using this method, all the irrelevant data contained in the environmental resources are fingerprinted together with the relevant information. We leverage the publicly available Rabin fingerprint implementation [39], and use the default 4 KB average chunk size.

Once the vendor produces its list of items for the environmental resources on the reference machine, it sends it to the user machines. Each user machine compares the list of items from the vendor to its own list and produces a new list with the set of items that differ from those of the vendor. The new list contains both items present on the reference machine but not on the user machine and vice-versa. This new list is sent back to the vendor, triggering the clustering algorithm.

Clustering Algorithm

The algorithm is divided into two phases. In the first phase, clustering is performed with respect to the resources for which there are parsers. Because for these resources we have precise information about the relevant aspects of the environment, two machines are assigned to the same cluster if and only if their sets of items that differ from the vendor are the same. We refer to the clusters produced in this first phase as "original clusters".

In the second phase, the environmental resources for which the vendor does not provide a parser are taken into account. In this phase, we need to decide whether to split the original clusters based on these resources. Since content-based fingerprints do not leverage semantic information about the resources, they are imprecise representations of the resources' contents. To cope with this imprecision, we use a diameter-based algorithm that is a variation of the QT (Quality Threshold) clustering algorithm [30] (we dismissed the traditional k-means algorithm because it is non-deterministic) on each of the original clusters.

Considering the user machines of only one original cluster at a time, the diameter-based algorithm starts with one machine per cluster. It then iteratively adds machines to clusters while trying to achieve the smallest average inter-machine distance, without exceeding the vendor-defined cluster diameter d. The distance metric is the Manhattan distance between two machines, defined as the number of different items associated with the resources for which there are no parsers. When the algorithm cannot merge any addi-

tional clusters, it moves on to the next original cluster. After the final set of clusters is produced, we also explicitly split clusters that contain machines with different sets of applications with overlapping environmental resources. The final clusters are labeled with their set of differing items.

Discussion

At this point, it should be clear that it is advantageous for the vendor to provide parsers for as many environmental resources as possible. Doing so allows the vendor to cluster machines more accurately. In addition, using parsers, the vendor can exercise greater control over the clustering process. For example, the vendor can specify which items it believes to be less important. The vendor can create bigger clusters by removing those items from the set of differing items of each machine. It is also possible to discard only a suffix of some of the hierarchical items. For example, assume that many machines have version 2.4 of libc, but a significant portion of them have a different library hash. This discrepancy probably means that there are machines that have version 2.4 of libc compiled with different flags. In deploying a non-critical upgrade to the graphical interface of Firefox, the vendor might decide to discard this information, thus putting all machines using version 2.4 of libc in one bigger cluster.

Relative to performance and scalability, the user-side fingerprinting and comparison processes are efficient and distributed, regardless of how many parsers are provided. However, producing as many parsers as possible has advantages in terms of the vendor's computational effort. The first phase of the clustering algorithm runs efficiently in time that is proportional to the number of user machines. In contrast, the second phase performs a number of comparisons that is quadratic in the number of machines in each original cluster, and each comparison takes time proportional to the number of items associated with resources for which there are no parsers. Thus, the second phase can become computationally intensive if there are many such resources. The running time of the QT algorithm is also quadratic in the number of machines in each original cluster. In our future work, we plan to develop an efficient incremental reclustering approach, since a relevant change in a machine's environment can change that machine's cluster.

3.3 Discussion and Current Limitations

Although the presentation above focuses on vendors and individual users, Mirage can benefit the administrators of large systems as well. In fact, a large system with uniform machine configurations is represented as a single machine in Mirage's deployment protocol. Even when system administrators are required to certify all upgrades manually before deploying them, Mirage can help them by keeping track of the applications that need to be tested, automatically testing each of them, and reporting unexpected behavior. Furthermore, by increasing the administrators' confidence in the upgrades, Mirage encourages administrators to apply upgrades sooner. In addition, by automating the reporting of failed upgrades, Mirage makes it easier for vendors to fix the problems with their upgrades.

Despite its many potential benefits, the current implementation of Mirage has some limitations.

The information that the vendor stores about the similarities between the user machines could be used by an attacker to quickly identify the targets of a known vulnerability. Although items contain hashes of the environmental resources' contents, the file names are currently stored in the clear. Computing a cryptographic hash of file names would not help, as the attacker could install the application and check for identical hashes of vulnerable files. However, we could enhance the security and privacy of our "original", parser-aided clustering, by letting each machine determine the cluster it belongs to locally based just on the comparison with the vendor's set of files and fingerprints and by communicating a single cryptographic hash of its items to the vendor. To enable staging, the vendor could transmit only the list of clusters' hashes to each machine. If necessary, machines could use some of the anonymity-preserving techniques [13] to communicate with the vendor. During deployment, the vendor could publicly advertise the cluster being tested currently and use only the number of nodes that belong to each cluster to determine when it is appropriate to move to the next stage.

3.4 Evaluation

We address the following three questions:

1. Can we accurately identify the environmental resources on which an application depends?

Application	Files total	Env. resources	False positives	False negatives	Required vendor rules
firefox	907	839	1	23	7
apache	400	251	133	0	2
php	215	206	0	0	0
mysql	286	250	0	33	1

Table 3.1: Effectiveness of the heuristic in identifying environmental resources.

2. Can we cluster machines into meaningful clusters such that members of a cluster behave similarly or identically with respect to an upgrade?
3. Can we use staged deployment to significantly reduce upgrade overhead with little or no impact on deployment latency?

Identifying Environmental Resources

We study a desktop application (Firefox) and three server applications (Apache, PHP, and MySQL). To determine false positives (files the heuristic erroneously flags as environmental resources) and false negatives (environmental resources the heuristic missed), we manually inspect the set of files produced by the heuristic, as well as their contents.

Table 3.1 shows, for each application, the number of files accessed in the traces, the number of environmental resources, the number of false positives and false negatives generated by the heuristic, and the number of vendor-specified rules necessary to obtain a perfect classification of the files.

The heuristic by itself is able to correctly classify a large fraction of the environmental resources. Combining the heuristic with a small number of vendor-specified rules correctly identifies all environmental resources, without false positives or false negatives.

For PHP, the heuristic correctly identifies all environmental resources, without false positives or false negatives.

For Apache, the heuristic erroneously classifies the access log and some HTML files from the root document directory as environmen-

tal resources – the log because it is accessed during initialization, and the HTML files because they are accessed read-only in each run.

For MySQL, the heuristic erroneously omitted the directory in which the `mysql` database is stored, because by default it excludes files from `/var`. These files, however, also contain configuration data and are part of the environmental resources of MySQL.

For Firefox, not all extension, theme, and font files are included properly by the heuristic, because Firefox loads them whenever they are needed (i.e., after the initialization phase). We had to specify the important file types so they are correctly classified.

As can be seen from the table, the number of files misclassified can vary significantly from one application to the other. The number of rules, however, stays very small because each of these rules recognizes one special case of misclassification under which a variable number of files may fall. For this reason, the number of rules required to obtain a perfect classification gives a better appreciation of the effectiveness of the heuristic in terms of its ability to classify files automatically.

Defining the rules is an easy matter of invoking the regular expression-based Mirage API. In general, however, some files and directories are located at different places on different machines. In this case, the vendor can easily provide a script to automatically extract the correct location of files and directories from relevant configuration files or environment variables and generate the regular expressions locally on each machine.

Clustering

As mentioned in Section 3.2, a clustering is evaluated relative to a particular upgrade and a set of problems. We start by introducing metrics that quantify the "goodness" of clustering. The first metric captures the number of unnecessarily created clusters (C), while the second metric captures the number of wrongly-placed machines (w), i.e., machines that behave differently than the rest of their clusters.

If there are p different problems, an *ideal* clustering creates $p+1$ clusters (a cluster for every problem, and a cluster containing all other machines), has no unnecessary clusters ($C = 0$), and has no misplaced machines ($w = 0$). A *sound* clustering has $C \geq 0$ and $w = 0$. With a sound clustering, therefore, multiple clusters may exhibit correct behavior or the same incorrect behavior. A clustering that is *imperfect* has $w > 0$ and potentially $C > 0$.

Machine Name	Description
fc5-ms4	Fedora Core 5
fc5-ms4/php4	Fedora Core 5 with PHP 4.4.6
fc5-ms4/php4/ap139	Fedora Core 5 with PHP 4.4.6 and Apache 1.3.9
fc5-ms4/php4-comments	Fedora Core 5, PHP 4.4.6 (comments in /etc/mysql/my.cnf altered)
ubt-ms4, ubt-ms4(2)	Ubuntu 6.06 (Dapper Drake) (two identical machines)
ubt-ms4/php4	Ubuntu 6.06 with PHP 4.4.6
ubt-ms4/php4/ap139	Ubuntu 6.06 with PHP 4.4.6, and Apache 1.3.9 (compiled with PHP support)
ubt-ms4/withconfig	Ubuntu 6.06 (added config. file /etc/mysql/my.cnf)
ubt-ms4/userconfig	Ubuntu 6.06 (added a user config file $HOME/.my.cnf)
ubt-ms4/confdirective-added	Ubuntu 6.06 (added a config. directive to /etc/mysql/my.cnf)
ubt-ms4/confdirective-deleted	Ubuntu 6.06 (deleted a config. directive from /etc/mysql/my.cnf)
ubt-ms4/comment-added	Ubuntu 6.06 (added some comments to /etc/mysql/my.cnf)
ubt-ms4/comment-deleted	Ubuntu 6.06 (deleted some comments from /etc/mysql/my.cnf)
ubt-ms4/libc-upg	Ubuntu 6.06 (upgraded to a new version of libc)
ubt-ms4/libc-upg/withconfig	Ubuntu 6.06 (added config. file /etc/mysql/my.cnf, libc upgraded)
ubt-ms4/libc-upg/userconfig	Ubuntu 6.06 (added a user config. file $HOME/.my.cnf, libc upgraded)
ubt-ms4/libc-upg/confdirective-added	Ubuntu 6.06 (added a config. directive to /etc/mysql/my.cnf, libc upgraded)
ubt-ms4/libc-upg/confdirective-deleted	Ubuntu 6.06 (deleted a config. directive from /etc/mysql/my.cnf, libc upgraded)
ubt-ms4/libc-upg/comment-added	Ubuntu 6.06 (added comments to /etc/mysql/my.cnf, libc upgraded)
ubt-ms4/libc-upg/comment-deleted	Ubuntu 6.06 (deleted comments from /etc/mysql/my.cnf, libc upgraded)

Table 3.2: Sample Linux configurations used in our experiments. All machines have MySQL 4.1.22.

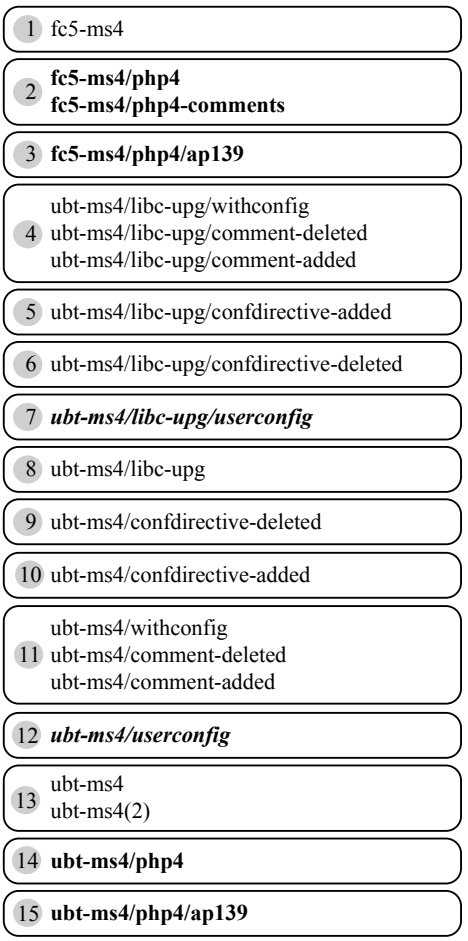

Figure 3.2: Clustering obtained by using parsers for all environmental resources. Entries in boldface experience the PHP problem with a MySQL upgrade, whereas entries in bold-italics will exhibit the my.cnf problem.

Unfortunately, an ideal clustering is typically not achievable in practice. Since a particular problem may affect only a subset of environments, there may be multiple clusters with machines that do not experience the problem. In contrast, sound clustering is achievable and useful in limiting the upgrade overhead, so it is the type of clustering that we seek to obtain.

We have reproduced a number of machine environments that exhibit real, non-trivial upgrade problems with two applications: MySQL and Firefox.

MySQL

Table 3.2 describes the machine configurations used in the MySQL experiment. On some of these configurations, the upgrade to a

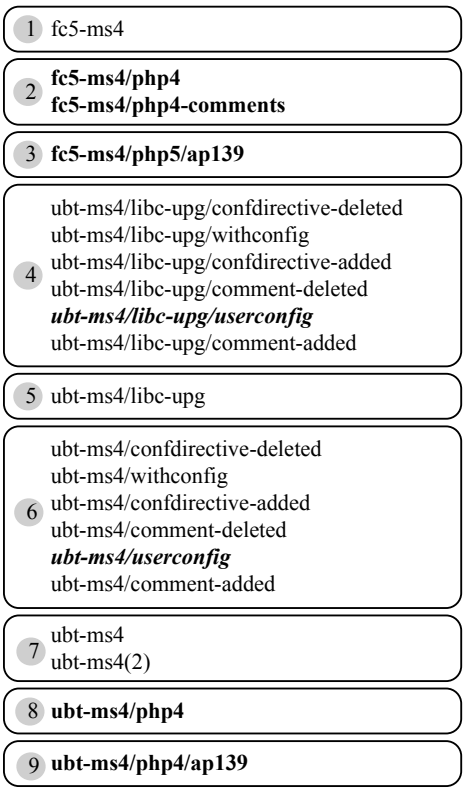

Figure 3.3: Clustering obtained by using only parsers supplied by Mirage and a diameter value of 3 for the remaining environmental resources. Entries in boldface experience the PHP problem, whereas entries in bold-italics will exhibit the my.cnf problem.

new version is successful. Other configurations (the ones with *php4* in their names) are known to exhibit broken-dependency problems with PHP [41] after a MySQL upgrade. Still others (the ones with *userconfig* in their name) exhibit legacy-configuration problems with the `my.cnf` MySQL configuration file.

Figure 3.2 shows the results of our clustering algorithm when we use application-specific parsers for *all* environmental resources. Entries in boldface denote machines that experience the PHP problem when upgrading MySQL, whereas the entries in bold-italics denote machines that experience the `my.cnf` problem.

As we see from Figure 3.2, the machines that experience the MySQL upgrade problems are not clustered together with machines experiencing other problems or machines behaving correctly, thus satisfying our main clustering goal ($w = 0$). During deployment, problems would be observed in only two clusters.

The results also show that the algorithm creates separate clusters for the different distributions, versions of libc, dependent applications, and differences in the configuration files. Although these differences have no bearing on the behavior of this particular upgrade, another upgrade could be problematic in one of these different environments. Thus, the algorithm must conservatively cluster these machines separately. Overall, this clustering created 12 additional clusters ($C = 12$) with respect to this upgrade.

We now examine the results of the clustering algorithm without using any MySQL-specific parsers; the initial phase of the algorithm relies on Mirage-supplied parsers only. As described in Section 3.2, the Mirage-supplied parsers deal with executables, shared libraries, and system-wide configuration files. Therefore, we have to execute the second phase of the clustering algorithm, which uses the QT algorithm for the resources that are fingerprinted with content-delineation. We show the final clustering results obtained with a diameter value of 3 in Figure 3.3.

The figure shows that, even without the application-specific parsers, our clustering algorithm correctly clusters the machines with the PHP-related problem. However, we see that clusters 4 and 6 both contain a machine that experiences a problem along with machines that do not. This clustering is thus not sound ($w = 2$) for an upgrade that triggers the `my.cnf` problem. This happens because the number of differences between the machines in this cluster is smaller than the diameter value. Specifically, the differences in this case are in the hash of the `my.cnf` file. Since the file is rather small and our Rabin fingerprinting takes hashes at the granularity of 4KB, only one hash is different. Using a diameter value of 0 would have separated the machines in this cluster, but it would also have separated machines that have benign differences (the machines with -comment in the name). In a real-world scenario with thousands of machines, this choice of the diameter could result in a large number of clusters that would slow down the deployment. This example shows that picking an adequate diameter value is difficult and that clustering might be imperfect, regardless of the diameter value chosen, when some parsers are missing.

The vendor can decide to vary the size of the resulting clusters depending on the upgrade that is to be deployed. For instance, in our setup we have 5 different types of changes to the MySQL configuration file `my.cnf`. If the vendor is providing a parser for this file and wants to deploy an upgrade that is unlikely to be affected by any of those changes, it can choose to ignore the corresponding

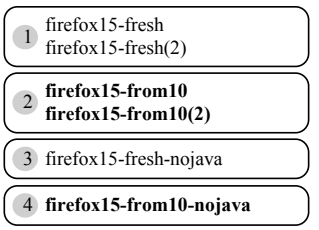

Figure 3.4: Firefox clustering obtained by using parsers for all environmental resources. Entries in boldface experience problems when upgrading to version 2.0.

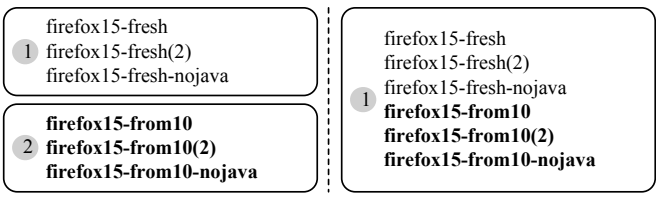

Figure 3.5: Firefox clustering results obtained by using only Mirage-supplied parsers. On the left, results with a diameter value of 4. On the right, results with a diameter value of 6. Entries in boldface experience problems when upgrading to version 2.0.

items. The resulting merge of clusters 4,5,6 and clusters 9,10,11 in Figure 3.2 can speed up the staged deployment. This regrouping of clusters still correctly separates the problematic configurations from other, "good" clusters. Such an effect cannot easily be achieved if the vendor does not provide a parser. Without a parser, creating larger clusters can only be done by increasing the diameter, which does not give any guarantee over how machines are going to be re-clustered.

Firefox

Table 3.3 describes the machine configurations used in the Firefox experiment. The three first configurations are fresh installations of version 1.5.0.7 of Firefox, with the third one having Java and JavaScript disabled. The other three configurations have been upgraded from version 1.0.4, and the last one has Java and JavaScript turned off. The latter three configurations exhibit a legacy-configuration problem when upgraded to Firefox 2.0. In particular, two preference files that existed in version 1.0.x (and were upgraded to 1.5.x) cause erratic behavior when Firefox is upgraded to version 2.0 [20].

Figure 3.4 shows results obtained using application-specific par-

sers for Firefox's configuration files, in addition to Mirage's supplied parsers. The clustering is sound ($w = 0$, $C = 2$) for this upgrade; there are two extra clusters with the two machines with Java and JavaScript disabled.

Figure 3.5 shows two clustering results using only Miragesupplied parsers. On the left, a diameter value of 4 is used. The problematic configurations are all in one cluster, and all the other ones in another cluster. For the upgrade we are considering, this clustering is ideal, with $w = 0$ and $C = 0$. On the right, a diameter value of 6 is used. This clustering is imperfect, as the problematic configurations are clustered with other machines ($w = 3$). These results show that picking the right clustering diameter is difficult and that a small difference potentially has a non-trivial impact.

The comparison between the clustering results in Figures 3.4 and 3.5(left) might suggest that it is sometimes better for the vendor not to provide parsers for its applications. However, this conclusion is incorrect. The reason is that, when the vendor is about to deploy an upgrade, it does not know if or where problems may occur. Thus, it would be improper to assume, as done in Figure 3.5(left), that machines with features turned off always behave the same as their counterparts with those features turned on. Firefox's configuration files contain many irrelevant parameters (such as timestamps and window coordinates) as well as relevant configuration settings (such as Java settings). Since diameter clustering only considers the number of differences, it is unable to make a distinction between relevant and irrelevant differences. If the vendor supplies the required parsers, it can discard the irrelevant differences to guarantee sound clustering.

Deployment Protocols

We still need to evaluate the impact of staged deployment on upgrade overhead and latency. Toward this end, out of the space of possible staged deployment protocols described in Section 3.2, in this section we evaluate two: one that seeks to front-load most of the vendor's debugging effort while reducing the upgrade overhead (i.e., the number of machines testing a faulty upgrade); and one that seeks to reduce upgrade overhead without disregarding the upgrade latency (i.e., the time elapsed until the upgrade is successfully applied and running on a machine). For both protocols, we assume that the representatives are always online and ready to fully test an upgrade, perhaps as a result of a financial arrangement with the vendor.

Machine Name	**Description**
firefox15-fresh, firefox15-fresh(2)	Firefox v.1.5.0.7 freshly installed (two identical machines)
firefox15-fresh-nojava	Firefox v.1.5.0.7 freshly installed, Java and JavaScript disabled
firefox15-from10, firefox15-from10(2)	Firefox v.1.5.0.7, upgraded from v.1.0.4 (two identical machines)
firefox15-from10-nojava	Firefox v.1.5.0.7, upgraded from v.1.0.4, Java and JavaScript disabled

Table 3.3: Configurations used in our experiments with Firefox. All machines run the same Linux distribution.

Front-loading the debugging effort while reducing the upgrade overhead.

The first protocol (called FrontLoading, for short) is divided into two phases. In the first phase, the vendor starts by notifying all representatives of all clusters in parallel that an upgrade is available. The representatives then download and test the upgrade using Mirage's user-machine testing subsystem (or another appropriate testing approach). Any problems during this phase are reported back to the vendor using Mirage's reporting subsystem. Once the problems are fixed by the vendor, all representatives are again concurrently notified about the corrected upgrade and can download, test, and report on it. This process is repeated until no more problems are reported by the representatives, ending the first phase of the protocol. No non-representatives test before the end of this phase.

During the second phase, the vendor proceeds by handling one cluster at a time, sequentially, starting from the cluster that is most dissimilar to the vendor's environment and proceeding (in reverse order of similarity) toward the cluster that is most similar to it. Ideally, no new problem should be discovered in this phase. However, because of imperfect testing or clustering, some problems may still be encountered. The vendor starts this phase by simultaneously notifying the non-representative machines of the first cluster in the order. The non-representatives of this cluster then download the upgrade, test it, and report their results. It is possible that the upgrade succeeds at some non-representative machines but fail at others. When this occurs, the non-representative machines that passed testing are allowed to integrate the upgrade, but are later notified of a new upgrade fixing the problems with the original upgrade. The machines that failed testing do not integrate the upgrade and are later notified of a (hopefully) corrected version of the upgrade. When there are no more failures and a large fraction (according to a vendor-defined threshold) of the non-representatives have passed upgrade testing successfully, the process is repeated for the next cluster in the order. The reason we only wait for a (large) fraction of the non-representative machines to report success is that some machines may stay offline for long periods of time; it would be impractical to wait for all these machines to pass testing before moving to the next cluster. When these "late arrivals" come back online, they are notified, download, test, and report on all the upgrades they missed.

This protocol front-loads the debugging effort in two ways: (1) it collects reports from all representatives of all clusters right from

the start, getting a broad picture of all the problems that can be expected; and (2) in its second phase, it proceeds in the reverse order of cluster distance from the vendor, as farther clusters are more likely to experience problems. These two characteristics also allow the protocol to reduce the upgrade overhead.

Reducing upgrade overhead with better upgrade latency.

In our second staged protocol (called Balanced, for short), deployment starts with the cluster that most closely resembles the vendor's installation, progresses sequentially to the next cluster that most closely resembles the vendor, and so on. The similarity between clusters and the vendor is evaluated by a distance function accounting for the number of differences in the environmental resources of machines belonging to the clusters. Each time the protocol progresses to a new cluster, the representatives of that cluster are notified about it in parallel and can then download, test, and report on it. This process is repeated until the upgrade passes testing successfully at all the representatives of the cluster. At this point, the non-representative machines of the cluster are notified and go through the same process. As in FrontLoading, non-representatives that pass testing successfully can integrate the upgrade, but may later receive a new upgrade. When all (or a significant fraction of) the machines in a cluster have successfully tested an upgrade, deployment progresses to the next cluster.

This protocol promotes low upgrade overhead by testing upgrades at the representatives of each cluster before allowing the much larger number of non-representatives to test them. Interestingly, it also promotes low latency for a large set of machines, as compared to our first protocol, in two ways: (1) by allowing many non-representatives to successfully pass testing before problems with all clusters are debugged by the vendor; and (2) by ordering clusters so that clusters that are more likely to pass testing receive the upgrades first. However, debugging at the vendor is more spread out in time, as compared to the first protocol.

Evaluation Methodology

The evaluation is based on an event-driven simulator of different deployment protocols and clustering schemes. The main simulator inputs are the number of clusters, the clusters' sizes, the clustering quality, the number of representatives per cluster, in which clusters the upgrade problems appear, and the times to download, test, and

fix an upgrade. Because our goal here is not to present an extensive set of results for a large number of simulator inputs, we select a particular scenario and discuss the corresponding results. The behaviors that we observe with this example scenario are representative of those of the other scenarios we considered.

Our example scenario has 100,000 simulated machines running Mirage. The times to download and test an upgrade are 5 and 10 time units, respectively. The time to fix each upgrade problem at the vendor is set at 500 time units. We choose the ratio between upgrade download and test vs. debugging time to mimic the ratio between the expected times in the field; download and test taking a few tens of minutes, with debugging (entire cycle at the vendor) taking at least one day. In all cases, we cluster machines into 20 clusters.

We consider two main categories of upgrade problems: prevalent (a large number of machines are affected by the problem) and non-prevalent (a smaller number of similar machines are affected). The results we show are for a case of one prevalent problem affecting 15% of machines (resembling the upgrade failure results reported in the literature [5]) and two non-prevalent problems in two different clusters.

We use two different clusterings. The first is sound with respect to the problems we defined; it has 16 more clusters than ideal clustering, with no misplaced machines. We create the second, imperfect, clustering by injecting a single misplaced machine in one of the clusters from the sound clustering setup. This machine is not a representative in the affected cluster. We assume that user-machine testing correctly identifies a problem if there is one, with no false positives. Hence, we assign one representative per cluster.

Given the lack of large-scale machine characterizations that we would require in this scenario, we assume that all clusters have the same size and study per-cluster, rather than per-machine, upgrade latencies. In the cases we consider, the vast majority of machines behave the same within a cluster. Hence, this setup enables us to discuss the qualitative features of our staged deployment protocols.

We compare our two protocols, FrontLoading and Balanced, against baselines called “NoStaging” and “RandomStaging”. The NoStaging protocol places all machines into a single cluster and treats them all as representatives to promote deployment speed at the possible cost of a high upgrade overhead. For this reason, NoStaging should be used for simple and urgent upgrades, such as security patches. The RandomStaging protocol resembles Balanced, but

the order of deployment to clusters is random. Although RandomStaging may not have any real use in practice, it does allow us to isolate the benefit of staged deployment from that of intelligent machine clustering. To evaluate this protocol, we create a scenario in which the problems are uniformly distributed across the deployment order.

Since we do not have an underlying set of machine environments, we consider two deployment scenarios for the Balanced protocol: 1) "best-case", in which the representatives discover problems at the latest possible time, producing the most favorable latency; and 2) "worst-case", with the problems being encountered early on, resulting in the worst latency. We compare the protocols in terms of their latencies and upgrade overheads.

Deployment Results

Most importantly, we expect our staged protocols to have significantly lower upgrade overhead than NoStaging. This benefit should come with little or no penalty in terms of upgrade latency.

Sound clustering. Our protocols indeed exhibit much lower upgrade overhead than NoStaging. With NoStaging, all machines that exhibit problems (m, in total) fail testing, whereas many fewer do so with the other three protocols thanks to their staged deployment (presumably, $m >> p$, i.e., the number of problematic machines is much greater than the number of problems, p). The main reason for this result is that staging forces the deployment to halt at the first cluster that exhibits a problem, sparing other machines from experiencing the same problem. Since the clustering is sound with no misplaced machines and we assume that user-machine testing finds the problem if one exists, the representative of each cluster with a problem experiences it. Hence, the total number of test failures is p for Balanced and RandomStaging. As expected, Balanced has lower overhead than FrontLoading. With FrontLoading, representatives of all clusters with machines exhibiting the prevalent problem ($p + C_p$ in total, where C_p is the number of additional clusters with problematic machines) fail testing.

Figure 3.6 shows the CDF of the upgrade latency, confirming our intuition about the protocols we consider. In NoStaging, 75% of the machines pass the upgrade test right away (at the cost of the high upgrade overhead we just discussed). In the best-case scenario, Balanced quickly applies the upgrade successfully at a large fraction of machines, significantly sooner than FrontLoading. Since

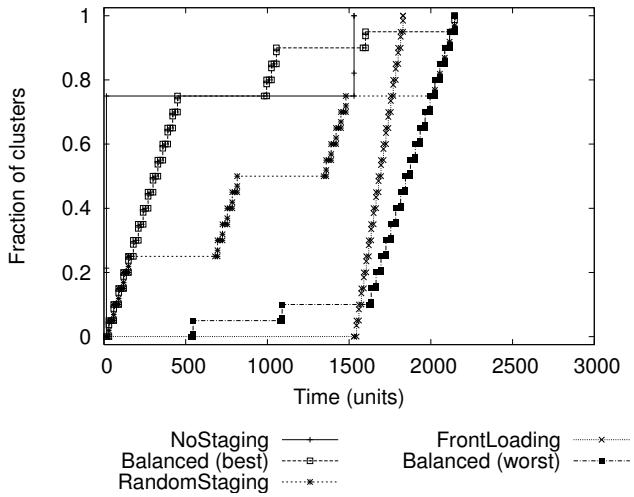

Figure 3.6: CDF of the per-cluster upgrade latency under sound clustering.

FrontLoading has an initial parallel testing and debugging phase, most successful upgrades are delayed by several debugging times at the vendor (500 time units each). Nevertheless, the last cluster applies the upgrade sooner under FrontLoading than the other staged protocols, especially when the number of clusters is large, since FrontLoading's second phase does not have an extra step of testing at representatives. RandomStaging's latency lies between that of the best-case and the worst-case scenarios for Balanced. We also see that additional clusters (when $C > 0$) slow down the sequential phases of the staged protocols.

Imperfect clustering. Figure 3.7 shows the CDF of the per-cluster latency when we position the misplaced machine (a problematic machine that behaves differently from the rest of its cluster) in the first or the last cluster in the order.

We see that FrontLoading can be further slowed down if the misplaced machine is in the first cluster. Arguably, this is in agreement with this protocol's main strategy. With its best-case scenario, Balanced is similarly slowed down in this case when the entire first cluster tests the upgrade and we encounter the misplaced machine. When the misplaced machine is in the last cluster in the order, this protocol is only marginally affected. Since it tests on all machines, NoStaging is not affected by the presence of the misplaced machine. Overall, the main trends among the protocols remain and the upgrade overhead is simply augmented by one extra machine that fails testing for all protocols.

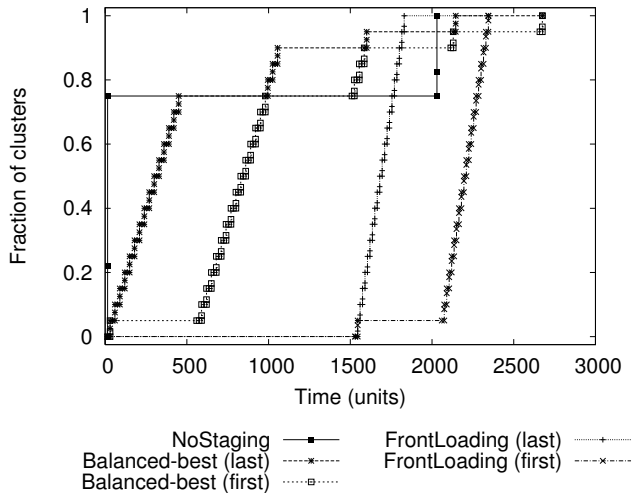

Figure 3.7: CDF of the upgrade latency under imperfect clustering. The position of the misplaced machine (its cluster in the deployment order) is shown in parenthesis after protocol names.

3.5 Conclusion

We introduce Mirage, a distributed framework that integrates deployment, user-machine testing, and problem reporting into the development cycle of software upgrades. The design of Mirage is motivated by our survey of system administrators, described in chapter 2, that points to a high incidence of problematic upgrades, and to the difference between vendor and user environments as one of the important causes of upgrade problems. We describe in detail two key innovations in Mirage: the clustering of user machines according to their environments and the staged deployment of upgrades to these clusters.

Our evaluation demonstrates that our clustering algorithm achieves its goal of separating machines with dissimilar environments in different clusters, while not creating an impractically large number of clusters. The algorithm performs substantially better when the vendor provides parsers for all of the environmental resources of the application. A simulation study of two staged deployment protocols demonstrates that staging significantly reduces upgrade overhead, while still achieving low deployment latency. Furthermore, a suitable choice of protocol allows a vendor to achieve different objectives.

Chapter 4

Concolic Execution for Testing Software Upgrades

4.1 Introduction

Deriving a correct software implementation is a difficult and arduous task. Programmers are often under time constraints to meet release deadlines, and therefore use a variety of static and dynamic analysis techniques to improve software reliability. In chapter 2, we presented a survey which demonstrates that software upgrades routinely ship with problems. To reduce the impact of those problems, in chapter 3, we described staged deployment, a technique that allows the vendor to deploy the upgrade in multiple stages, detect problems early, and reduce the exposure of users to these problems.

Of course, this technique only makes sense after the vendor has gained enough confidence that most problems have already been eliminated. Even with staged deployment, the vendor needs to test the application up-front, requiring suitable techniques and tools. Then, during staged deployment, users (and in particular cluster representatives) must again test the upgrade and report problems. Since users in general do not have the skills necessary to test an application, it is necessary to provide a fully automated solution.

Static analysis [4, 6] can automatically check many useful properties, e.g., whether the API is being used properly. Unfortunately, static analysis can be incomplete and miss many important problems, or it can overwhelm the tester with a large number of spurious warnings. For user-machine testing, it is inadequate, as it is too abstract and fails to take into the account the actual environment in which the application is running.

A larger class of programming errors can be identified using dynamic analysis or testing [40]. Any mature program has a test suite

that has been put together over time. A classic goal for testing is adequate code coverage. Ideally, testing should cover each statement of a program at least once. This seemingly simple task is complicated by the exponential explosion in the number of paths, resulting from branches in the source code. Many tools can maximize the code coverage achieved by a set of inputs [53]. Unfortunately, achieving full coverage is difficult, even for mature code with several decades of development [10]. Furthermore, manual test generation is a labor-intensive task and requires significant investment from the software vendor. For user-machine testing, it is difficult to use test cases and make sure that they are compatible with the user environment. For instance, a test case may exercise a feature of the application that is not meant to work given the current configuration of the application at the user site, and may therefore trigger a false positive.

An approach for comprehensive, automatic test generation that has gained considerable attention recently is symbolic execution [10, 11, 27]. The goal of symbolic execution is to systematically explore all possible paths in a program, looking for conditions that may lead the program to crash (e.g., de-referenced null pointers, divisions by zero, off-by-one array accesses, and failed `asserts`). In more detail, a symbolic execution engine marks all program input as symbolic (i.e., having arbitrary value), and then runs the code while propagating the symbolic inputs to program variables. When it encounters a branch, the engine queries a constraint solver to compute the conditions that can lead to the two sides of the branch, updates the cumulative constraints for each side, and takes both sides in parallel. The engine proceeds to systematically cover all branches and explore all paths, if sufficient time is available. As it traverses the entire program, the engine also creates and explores constraints that check for potential crash situations.

A variation of symbolic execution, called concolic execution [9], has the same full-coverage goal but explores paths in a different manner. In concolic execution, the program is first executed with a random set of inputs, e.g., all zeros. Every time the program executes a branch, the concolic execution engine records the constraints on the input that led to the taken side of the branch. It also records constraints that check for possible crashes. After execution with this input is completed, the engine has a list of constraints. It then negates, one at a time, each of these constraints, and invokes the constraint solver to find a set of inputs that satisfies the constraints. The engine then re-executes the program, using as inputs the values selected by the solver. As before, it records the constraints it finds.

The process repeats until all paths are explored or the available time expires.

Although promising, these approaches have several problems. Most obviously, the number of paths to explore is usually extremely large, typically exponential in the size of the code. Several heuristics can be used to try and optimize the code coverage given a certain time budget, but those heuristics tend to get overwhelmed and get stuck in various places of the code. The consequence is that symbolic (or concolic) execution currently does not deliver on the promise of providing thorough, exhaustive testing. Instead, it provides only partial coverage of potentially uninteresting parts of the code. This problem is further exacerbated when doing regression testing. In this case, running symbolic or concolic execution is not ideal, as it always starts testing the application from the beginning and will therefore test over and over again the same parts of the code instead of focusing on the new or modified code.

In this chapter, we argue that it is necessary to develop new techniques to make symbolic execution suitable for regression testing of software upgrades. We present Oasis, a state-of-the-art concolic execution engine designed as the testing component of Mirage. Oasis can be used for up-front testing at the vendor or for integration testing on user machines. In Oasis, we experiment with novel techniques to enable focused testing of the new code in a software upgrade. In particular, we extend state of the art concolic execution [28, 10] in the following ways:

- Oasis implements a new search heuristic which uses a combination of static and dynamic analysis to identify the parts of the code modified in, or affected by, the upgrade. It then steers the exploration to cover those parts earlier and more intensively. This heuristic can be used at the vendor for up-front testing, or on the user machine to detect integration problems.

- We propose interactive symbolic execution, a new technique allowing the vendor to understand and influence the exploration of the code through a graphical user interface. This enables development of better search heuristics or interactive guidance to manually force the concolic execution engine to explore interesting parts of the code.

- For user-machine testing, Oasis leverages existing input on the user machines (such as configuration files) to avoid spending a large amount of time exploring input parsing code.

Oasis includes an optimization for testing upgraded code. Specifically, Oasis pinpoints the differences between the old and the new versions of the program (hereafter called "new code"), and prioritizes the exploration of paths and constraints relating to new code. During exploration, each time Oasis can choose to follow a specific side of a branch, it uses an inter-procedural control flow graph to compute the distance between the branch and the closest piece of yet uncovered new code. The distance is expressed as the number of branches (i.e., nodes in the cfg) and represents the expected difficulty in reaching this part of the code. Oasis thus systematically schedules the next path for exploration that is the closest to yet uncovered new code. During execution of each path, Oasis dynamically tracks the effect of the new code on the rest of the code. This way, it identifies regions of the old code who would have behaved differently had the new code not been executed. When scheduling paths for exploration, if several branches are at equal distance from the next block of uncovered new code, Oasis favors branches who themselves have been affected by the execution of new code. This heuristic allows Oasis to more thoroughly test the new code, which is the most likely place to find bugs when regression testing. Other engines (symbolic or concolic) do not explicitly target new code and may waste precious time exhaustively exploring parts of the program that have already been tested in previous versions.

Along the lines of trying to cover hard-to-reach parts of the code, previous work has demonstrated that using existing test cases to bootstrap the exploration of the code allows symbolic execution to reach deep parts of the program much faster [28]. We extend this by leveraging existing input when doing integration testing on user machines. Oasis starts the execution of the program and uses concrete input for each configuration and systems file. The list of those files can be specified by the developer, or inferred by Mirage (see chapter 3). For regular input (i.e., not coming from the environment), Oasis uses symbolic memory and proceeds as a normal concolic engine. If available, Oasis uses previously recorded concrete input. For instance, some representatives (such as system administrators in companies) may want to record specific input scenarios to test important features of an application after an upgrade. Oasis provides infrastructure so simplify recording of the input and uses it to speed up testing. Leveraging the input from the environment and the recorded scenarios, Oasis quickly passes input validation and is able to test deep paths in the application. Other engines (symbolic or concolic) would have difficulty reaching these deep paths within

the same time budget.

In some cases, even with the help of concrete input, existing test cases, and fancy heuristics, it may be difficult to cover the interesting parts of the code (such as the new code) in a reasonable amount of time. Others have experimented with state-merging [7] or using higher level constraints solver [26, 33] to address the problem of state explosion and improve coverage. While those techniques are very interesting and promising, much research is still needed before they scale and become practical for large software systems. In this dissertation, we propose a completely different and new approach called "Interactive symbolic execution". Instead of trying to improve the technology to scale symbolic execution, "interactive symbolic execution" exposes the problem of path explosion to the tester. Through a graphical user interface, the tester can visualize the state and progress of the symbolic exploration. Each path is displayed with its input, set of symbolic constraints and coverage statistics. Interactive symbolic execution is very useful to understand how the heuristic progresses and develop new ones tailored to the particular program or testing goal at hand. Furthermore, the tester can decide to interactively guide the exploration, for instance by changing the priority of some paths or canceling others. This allows the tester to manually drive the exploration towards interesting parts of the code.

When Oasis detects a problem, it automatically produces a detailed report including the input necessary to reproduce the bug. This considerably simplifies the problem of debugging for the vendor. When used for user-machine testing, Mirage stops deployment of the upgrade as soon as Oasis detects a problem. Oasis then sends the bug report to the vendor. While reporting the complete input is likely to leak privacy, we assume here the existence of business agreements between the vendor and the representatives that guarantee confidentiality. In chapter 5 we describe another technique to report problems in a way that better preserves privacy.

We present a preliminary evaluation of Oasis and its novel techniques: the heuristic to drive exploration of the code towards new or affected code in the upgrade, and interactive symbolic execution. Our results suggest that our heuristic is effective in more intensively testing upgraded code, and that interactive symbolic execution is a useful addition to understand and complement exploration heuristics in symbolic execution.

The contributions of this chapter are the following:

- The design and implementation of Oasis, a state-of-the-art

```
1  main() {
2  /* Memory for 200 config parameter strings,
3     each max of 80 chars long */
4    char configVars[200][81];
5    for (int j = 0; j < 200; j++ )
6      strcpy (configVars[j], "");
7
8    int cf = open(configfile);
9    int i = 0;
10   while (!EOF(cf) && i < 200) {
11     char* r = readline(&configVars[i], cf);
12     if (!r || !isValid (configVars[i])) {
13       exit(-1);
14     }
15     i++;
16   }
17
18 /* Buffer to read file contents */
19   char* buffer;
20   buffer = (char*)malloc(sizeof(char)*1024);
21   do_something(buffer, configVars);
22   free(buffer);
23
24 /* The printf is an invalid mem reference */
25   if (!strcmp(configVars[10], "value")) {
26     printf("\%c", buffer[someindex]);
27   }
28 }//end main
```

Listing 4.1: This example bug demonstrates the difficulty that concolic executions have in discovering bugs "deep" in the code and how execution with valid inputs can alleviate the problem.

concolic execution engine supporting heuristics to effectively test software upgrades.

- A new heuristic to drive symbolic execution towards new or affected code in an upgrade.
- A new approach called Interactive Symbolic Execution allowing the tester to interactively understand and influence the exploration of the program in a symbolic execution engine

The rest of this chapter is organized as follows. Section 4.2 provides additional background and motivation. Section 4.3 details the design and implementation of Oasis. Section 4.4 presents our preliminary evaluation. Finally section 4.5 concludes the chapter.

4.2 Overview

We motivate the design of Oasis using the example in Listing 4.1. This listing is a small variation of a real bug in the $ssh - keysign$

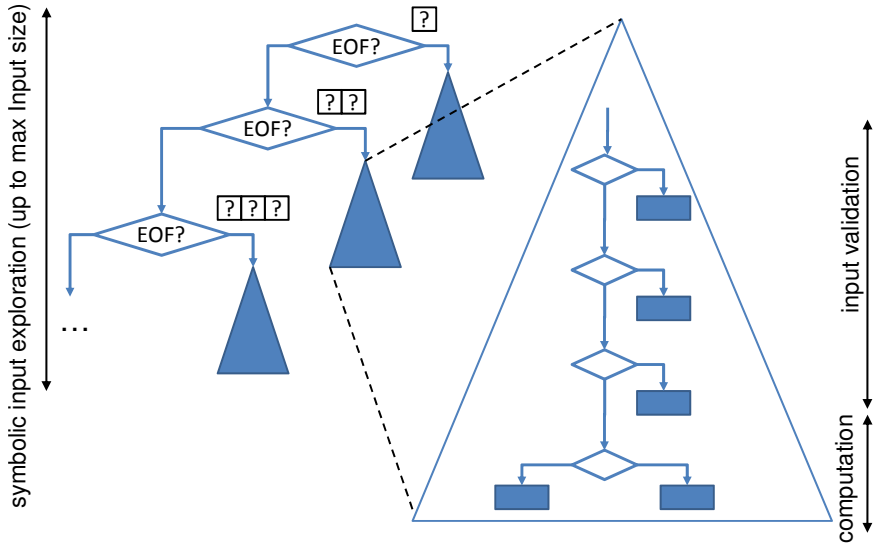

Figure 4.1: Path exploration by a conventional concolic execution engine on typical code. EOF in this figure refers to the check for the end of file or any other way of limiting input. Boxes with question marks refer to symbolic input that grows in size during path exploration.

tool of OpenSSH version 3.0 [44]. The code reads a line at a time from the configuration file and validates it (lines $8-16$). When this is done, it allocates a buffer, uses it (line 21), and frees it (line 22). However, under a very specific condition, it tries to print the now freed buffer (line 26).

We now describe how a conventional concolic execution engine would handle this program. The first input to the program is the configuration file. We assume that this file has 1000 bytes, for example. Concolic execution starts execution with all inputs set to a random value, e.g., all zeros. Thus, the entire configuration file would have 1000 bytes of zeros. We then start execution of the program with this input. The function readline (line 11) reads the input until it sees a line termination character (LTC) or until it has read 80 bytes, whichever comes first. With the given input, readline would read 80 zeros, which would fail to validate, and the program would exit. At that point, the execution would have generated 80 constraints of the form $byte_i \neq LTC$, where $0 \leq i \leq 80$. The engine would then negate each of these constraints in turn, causing it to take an input sequence that does not have 80 zeros as its first 80 bytes. For instance, it could try a 1 followed by 79 zeros next. With this input, the same sequence of events would occur, and so on. It is easy to see that the concolic engine would spend an enormous

amount of time, simply exploring the very first part of the program, where input is read and validated. It would do so until it guessed a set of string values that pass the validation tests, and, until it did so, it would never reach the code past the input validation, and therefore it would not be aware of the code that causes the bug. Figure 4.1 illustrates path exploration by a conventional concolic execution engine on our example code.

Let's now consider the situation where we have a number of valid configurations from a test suite or from an actual user environment. For simplicity, assume we have just one such configuration file. Oasis leverages this input as we described in the previous section. It would start executing with this configuration file as input, pass the input validation, buffer allocation and free, and arrive at the if statement on line 25. Most likely, the if condition would not hold, and Oasis would exit without accessing the buffer in the body of the if statement. In this execution, Oasis would have collected constraints on each byte of input during the validation phase, a certain number of constraints in the `doSomething()` function, plus a constraint stating that the if condition on line 25 is false.

In the next step, Oasis would now negate one of these constraints. Let's say that our strategy is to negate them one by one, starting from the first recorded one. This would cause us to pick an input different from the one we had on the first line of the configuration file, and most likely fail input validation. The same would happen for all the constraints related to the input variables and the `doSomething()` function, but then we would hit on the the constraint that says that the if condition was false. We would negate this one, find an input, and this one would trigger the bug.

Exploring this line of thought a little further, assume that the `printf` was a new line of code that was added as the result of a new version. In that case, since it is the closest to the new code, we would negate the condition right before `printf` and immediately find the problem. A concolic engine could not have found this bug so soon, because it would have had trouble generating valid input and then would spend a large amount of time exploring the input validation code and the code in `doSomething()`.

In a more general case, the tester may be interested in testing specific parts of the code, whether because they are part of the new version, or simply because they have not been tested enough yet. In that case, given the high number of constraints collected and the resulting number of possible paths to explore, with any given heuristic, the tester is left waiting and hoping that those parts

are going to be explored by the concolic engine. An alternative is to manually write test cases, but this is tedious and difficult. With interactive symbolic execution, Oasis would expose the list of constraints yet to be explored, along with possible input solutions and computed distances to uncovered code in the control flow graph. Let's say that the tester is more interested in testing the code in `doSomething()` than the input validation code. The tester would be able to identify the corresponding constraints easily and ask the engine to explore them immediately. Alternatively, the tester could decide to remove some or all of the constraints related to parsing the input and therefore allow the engine to focus on the `doSomething()` constraints immediately.

This style of program is quite common: an extensive first phase in which the input is read, parsed and validated, followed by a second phase in which the program performs its real function with inputs that have been validated. Web servers are another example. They read a large configuration file, before they start accepting incoming requests. Even in the handling of an individual request, the input string is first parsed and validated, before the requested operation is executed.

It is this observation that motivated the use of valid inputs to start Oasis execution along with a heuristic directing exploration towards new code and "interactive symbolic execution". The heuristic and interactive symbolic execution are two different ways to keep the exploration focused on interesting parts of the code, and using existing input Oasis can proceed past the input validation phase and test the part of the program in which it does its real work. With regular concolic execution, the engine is liable to get bogged down for a long time in testing less relevant parts of the code.

We want to stress that we do not argue that Oasis is better than regular concolic execution in some absolute sense, in terms of finding more bugs, or finding them more quickly, or providing much better coverage. Rather, we argue that we are able to explore certain parts of a program much sooner than a regular engine, in particular new parts of the code that are deep in the execution.

4.3 Design and Implementation

Oasis Overview

We have derived Oasis through extensive modification of the Crest open-source concolic execution engine [9]. Oasis instruments C programs using CIL [42]. The instrumentation is used to propagate

symbolic information and to flag crash-prone statement (i.e., statements such as assertions, divisions, memory reads and writes). Oasis incorporates many of the features of current state-of-the-art concolic and symbolic execution engines [10, 11, 9].

Programs being tested with Oasis always run with concrete inputs. This makes interaction with the external environment simple: programs running with Oasis can interact with outside libraries or the filesystem. However, in order to be able to track symbolic constraints on input that is passed to external libraries, Oasis needs visibility into the library functions. This is achieved by either recompiling those libraries with the instrumentation, or by providing models. Oasis provides its own instrumented version of *uclibc*, a simple *libc* implementation. In addition, Oasis provides a set of functions modeling interaction with the filesystem and the network. This allows Oasis to consider input coming from these sources as being symbolic (i.e., to log the constraints imposed on this input), and to support system calls operating on symbolic memory.

To generate inputs exploring different paths in the program, Oasis uses an off-the-shelf constraint solver, called STP [22]. Because the running time of the solver can be quite high, Oasis tries to reduce the complexity of the query to the solver by determining the minimal set of dependent constraints that need to be solved. To reduce the number of queries to the solver, Oasis caches previous solutions from the solver.

When running with previous test cases, Oasis maintains an tree of constraints, called the exploration tree. This allows it to avoid exploring several times the same path, for instance when several test cases share an identical prefix of constraints.

Oasis takes advantages of multi-core architectures by running the exploration in parallel on multiple-threads. A main scheduling thread is responsible for running the exploration heuristic and forking off path to be explored in parallel. A set of worker threads are responsible for invoking the solver and running the program under test.

What is Symbolic?

Oasis tracks symbolic constraints on the input, therefore it needs to know what in the address space of the program is input. There are different ways of marking input as symbolic. By using the $-symbolic\_argv$ parameter of the *oasis* command, the user can specify the number of symbolic command line parameters, along with their size. By inserting calls to $oasis\_sym(address, size)$ within

the program, the user can mark a specific address as the beginning of symbolic input of size *size*. This is very helpful to mark input coming from uninstrumented libraries.

Collecting Symbolic Constraints

Oasis instruments the program under test using CIL[42]. For each statement in the program, the instrumentation adds a few calls to a library that is used to keep track at runtime of the statements executed in the program and record the symbolic constraints. The way Oasis instruments the program is inherited from Crest[9] and extended in order to allow tracking for bugs. We therefore only describe here the main aspects of our approach and refer the reader to the literature for further details.

During execution, Oasis maintains a map of addresses to expressions, representing the address space of the program, hereafter named symbolic address space or SAS. An expression in Oasis is a tree composed of symbolic variables, constants, memory objects (arrays of expressions), and operators. Whenever input is marked as being symbolic, Oasis inserts in the SAS, at the address of the real input, one symbolic variable per byte of input. Oasis records in the SAS an expression composed of a memory object for each array[1] which is allocated (either statically or dynamically) in the program. Similarly, Oasis deletes memory objects from the SAS, when their corresponding array is freed. When the program executes a binary operation, such as an addition or a subtraction, Oasis maintains the corresponding binary expressions in the SAS.

We refer to an expression as being symbolic if, somewhere in the tree, the expression references a symbolic variable. In other words, an expression is symbolic if it involves the input of the program.

While maintaining the symbolic address space, Oasis logs the symbolic constraints corresponding to control flow in the program. For instance, when a conditional branch is executed, and the expression corresponding to the condition is symbolic, a new constraint is logged. If the program takes the *true* side of the program, the constraint logged is the symbolic expression. If, however, the program follows the $false$ side of the branch, the constraint logged is the negation of the symbolic expression.

Before letting the program execute any crash-prone statement, such as a division or a memory read/write, Oasis checks that the statement is not going to crash the program. For divisions, it checks

[1] We handle C structures and buffers similarly.

that the divisor is not zero, and if it is, it reports a bug. Otherwise, if the divisor is a symbolic expression, Oasis logs a constraint stating that the divisor was not zero. Similarly, for memory reads or writes, Oasis checks that the address being accessed points to a memory object and is within the bounds of this object. If this is not the case, Oasis reports a bug. If it is, and the memory accessed is symbolic, Oasis logs a constraint stating that the offset is within the range of the memory object (i.e., $0 < offset < memory\_object\_size - 1$).

Oasis handles reads and writes to memory through symbolic pointers in the same way it handles reads and writes to arrays and structures. Oasis supports symbolic pointers to pointers. For each access, Oasis logs a constraint stating the current value of the symbolic pointer. During path exploration, it then enumerates every possible location to which the symbolic pointer to pointer can point.

Handling System Calls

In a way similar to KLEE [10], Oasis provides a set of system call wrappers that are able to return either symbolic memory or concrete input, depending on whether the user has decided to treat a certain file or socket as symbolic input or not. The user can decide how many symbolic files will be used and their maximum size.

System calls do not always behave deterministically. For instance, when executing the `read()` system call, the kernel can decide to return between 0 and the amount of memory requested. For this reason, programs are required to check the return value of `read` and behave accordingly. To allow exploration of those paths, Oasis provides an option to turn the return value of those system calls symbolic. When this option is enabled, Oasis associates a symbolic variable and a constraint to the return value of each system call, allowing subsequent paths to explore different values. This is particularly helpful with some system calls, such as `read` or `select`, which can return any combination of file descriptors ready for I/O.

Constraint Solving

To drive the exploration of different code paths, after each execution of the program, Oasis negates one of the logged constraints and invokes STP to try to find a set of input values satisfying the symbolic constraints.

Because of the way constraints are modeled in Oasis, the translation of those constraints into STP is straightforward. Each symbolic

variable is translated into an 8-bit STP bitvector, memory objects into STP arrays, and binary operations into STP operations.

During an execution, Oasis routinely collects hundreds or even thousands of constraints. To explore a different path, Oasis negates one of those constraints and seeks to find a suitable input satisfying the negated constraint and all the previous constraints. The time needed for the solver to find a solution can be considerable. Fortunately, the negated constraint most of the time only involves one or a few symbolic variables, which are themselves involved only in a few other constraints. Thus, it suffices to query the solver for a solution to the set of constraints involving the variables appearing in the negated constraint. For example, consider the following set of recorded constraints: $\{x + y = 10; y < 10; z = 3\}$. Solving the constraint $z \neq 3$ does not require solving the two first constraints. We therefore only query the solver for the constraint $z \neq 3$, and reuse previous values of x and y. Oasis inherits from Crest its ability to compute subsets of dependent constraints before querying the solver.

Constraint Cache

Despite trying to optimize the query to the solver, the time needed for the solver to find a solution can still be considerable.

To avoid expensive calls to the solver, state-of-the-art concolic (and symbolic) execution engines implement a cache of solutions. In Oasis, we use a simple table mapping a set of dependent constraints to the corresponding solution. For instance, reusing the example of the previous section, assume that we collect constraints $\{x + y = 10; y < 10; z = 3\}$ with solution $\{x = 4; y = 6; z = 3\}$, after executing the program. We insert two entries into the cache: one lists the constraints $\{x + y = 10; y < 10; \}$ mapping to the solution $\{x = 4; y = 6\}$, whereas the other lists the constraint $\{z = 3\}$ mapping to solution $\{z = 3\}$.

Because programs frequently re-execute the same parts of the code with different inputs, they impose the same constraints on different symbolic variables. Oasis uses this observation to increase hit rates. Specifically, before inserting a solution for a set of dependent constraints in the cache, we rename the symbolic variables in the constraints in the order in which they appear. Returning to the previous example, we rename the variables x and y in the constraint $\{x+y = 10\}$ to $\{v_0+v_1 = 10\}$. If we ever encounter another constraint of the same form, but referencing different variables, we quickly find a solution in the cache.

Running with Actual Inputs

Oasis leverages any valid input available in order to explore deeper and longer code paths sooner than traditional concolic engines. The input can come from a test suite or from actual files present on the machine on which Oasis is running.

To take advantage of these valid inputs, Oasis needs to know the constraints resulting from the execution of the program with those inputs. This is done by running the program within Oasis in "recording" mode, and feed the input normally to it. In recording mode, the models of the filesystem and the network described in Section 4.3 operate as simple pass-through, in which they read from the filesystem or the network normally and tag the input as being symbolic.

This has to be done only once for each set of inputs. Oasis saves the inputs and the constraints to disk, and reuses them whenever the program needs to be tested again. Of course, if a new version of the program is to be tested, then the new version needs to be executed with the old inputs, and new constraints collected. Oasis does that automatically.

When using Oasis for user-machine testing, only some files are used with concrete values, the rest of the input is symbolic. In this case, no recording is required. An environment variable is used to tell Oasis which files will be used concretely (for instance, all files in */etc*). All files specified by this variable will be treated normally and the system call models will act as pass-through to the actual system calls.

Prioritizing new Code

When doing regression testing, Oasis takes advantage of the code differences between the old and the new version of the program. It uses this information to prioritize the exploration of paths and constraints that are affected by new code.

In the current version of Oasis, we use a very simple code differencing scheme: we simply compute a (textual) diff between the old version and the new version. More sophisticated approaches could be used, e.g., relying on information from a program development environment. These techniques could do a better job of identifying the precise differences, but the principle of how to use the information in Oasis remains the same.

During an Oasis execution, when a statement creates or manipulates an expression, Oasis checks if the statement is present in the

diff. If so, it flags the symbolic expression as being "new". This new flag is propagated to every expression in which a new expression is referenced. The propagation continues until the expression is logged in a symbolic constraint, which is then also flagged as being new. Such new constraints will be given priority in the exploration of different paths, as explained below.

To direct the search towards new code, Oasis builds an inter procedural control-flow graph of the program (hereafter called cfg). This graph is a binary tree with each node representing a branch in the program and edges representing the code in between branches. After each program execution, Oasis updates coverage statistics by marking which nodes and edges of the cfg have been covered. It then computes the distance between each explorable constraint (branch) and the closest node or edge in the graph which has not been covered yet. It also computes the distance to the next uncovered node or edge which is part of the new code. The distance is expressed as the number of nodes in the cfg that need to be crossed before reaching the target. The distance metric is therefore inversely proportional to the likelihood of reaching the targeted code by exploring the source branch.

Search Strategy

Listing 4.2 shows pseudo-code for the heuristic that Oasis implements. There are three important data structures: the CACHE that holds solutions to constraints obtained in earlier executions, CONSTRAINTS which is a sorted list of constraints yet to be explored, and a QUEUE that holds the next constraints scheduled for exploration.

During initialization, Oasis executes the program under test with random input first, then with each set of the valid inputs (if available), inserting the results in CACHE and CONSTRAINTS. The worker threads and the main thread are subsequently started.

In the main thread, Oasis keeps the CONSTRAINTS list sorted according to the heuristic, and pushes work to the worker threads' QUEUE. When a worker thread finishes his job, the main thread updates the coverage statistics, resorts the CONSTRAINTS and pushes more work to the QUEUE.

Each worker thread waits until there are constraints in the QUEUE. It then pops one, negates it, finds a solution for the negated constraint (either from the cache or by invoking the solver), executes the program with the new set of inputs, and finally inserts the new

execution, with its inputs and constraints, in CACHE and CONSTRAINTS.

The constraints are sorted for exploration in the following order. First, we examine constraints that involve crash-prone operations (those that may crash the program) coming from new code. Second, we consider the remaining constraints that involve crash-prone operations. Third, we examine constraints that are at the closest distance from yet uncovered new code. If several constraints are at the same distance, constraints affected by the new code are favored. Fourth, we pick constraints that are affected by the new code. Finally, we pick constraints that are at the closest distance from uncovered code.The idea behind this ordering is to drive the search first towards new code that can potentially cause bugs, then to new code, and finally to other code. Within each of these five priorities, we look at the constraint in the order that they were recorded in the program. Since Oasis starts execution from valid input, it already reaches parts of the code deep in the execution. Exploring constraints in the order in which they appear allows to explore those paths with more breadth.

Interactive Symbolic Execution

Oasis exposes to the tester its main data structures through a graphical user interface. Figure 4.2 shows a screenshot of the main window. This interface is built using the QT toolkit and presents windows and buttons allowing to understand and interfere with the exploration. This is particularly helpful when using Oasis for upfront testing, for instance, by the vendor.

The main window exposes the list of sorted constraints. For clarity, those constraints are displayed as a tree. Each top level entry displays the informations for a branch in the code, such as its location in the source code, the distance to the uncovered code and the distance to uncovered new code. Second level entries represent the instances of the branches that have been discovered during exploration. For each branch in the code, there may be from a few to several hundreds of instances collected during the various paths explored.

A second panel displays the scheduling decisions and whether the worker threads were able to solve the particular constraints and run with input. This allows the tester to visualize the progress of the heuristic and to realize if Oasis is currently stuck exploring a large number of uninteresting branches.

Because Oasis keeps running while the tester inspects its behavior, it may be difficult for the tester to make a decision before the state of Oasis changes. Therefore, a button allows the tester to put the exploration on hold while he decides on the best course of action, at which point clicking on the same button resumes exploration.

The tester can select one or more constraints in the list of constraints and decide to kill them. This instantly removes them from the CONSTRAINTS data structure. Because Oasis keeps exploring new paths, it is possible that new instances of the constraints that were killed are discovered again in subsequent executions. In order to avoid having the tester repeatedly kill constraints from the same branches over and over again, Oasis allows the tester to insert them in a blacklist. This way, newly discovered constraints for blacklisted branches are discarded immediately and do not clog the CONSTRAINTS queue anymore.

Finally, the tester can decide to schedule some constraints immediately, in which case the selected constraints are pushed to the QUEUE and explored by the worker threads.

The graphical user interface allows the tester to drive Oasis's exploration in a focused way. Instead of letting the heuristic spend a large amount of time in potentially uninteresting places of the code, the tester can decide to interfere with it and optimize the coverage. While very useful when regression testing an upgrade, this can also be used for other purposes, such as for instance debugging, when the developer suspects the area of the code in which the bug lies but does not know exactly where.

4.4 Evaluation and Discussion

In this section, we present a preliminary qualitative evaluation of the novel aspects of Oasis. We leave a more thorough, quantitative evaluation for future work.

We run Oasis with two versions of the `uServer`, an open-source Web server with approximatively $32K$ lines of code. We configure Oasis to provide the following input to the program: between 0 and 3 arguments of up to ten bytes of symbolic memory, and between 0 and 2 sockets returning up to 50 bytes of symbolic memory.

Prioritizing New Code

To compare our heuristic prioritizing new code (hereafter called `diff search`), we implement a heuristic optimized for code coverage and inspired by the one used in KLEE [10]. This heuristic

is very similar to `diff search` except that it ignores all the priorities related to the new code. It therefore always seeks to schedule branches which are at the smallest distance from the next uncovered block of code. We call this heuristic `cover search`. In addition to `cover search`, we also compare with a simple depth first search approach (`dfs`).

Figure 4.3 shows the code coverage obtained with the three configurations of Oasis. As can be seen from the figure, `diff search` and `cover search` end up covering the same amount of code after 30 minutes of exploration. `DFS` performs slightly worse, which is surprising considering that the literature describing similar techniques shows much greater benefits over `DFS`. This can be explained by the fact that the `uServer` is a complicated program processing highly structured input. It is therefore difficult for any strategy to make significant progress. In this example, `diff search` is actually the fastest strategy to reach the maximum code coverage achieved. This result shows that prioritizing new code does not necessarily penalize code coverage in general.

Figure 4.4 shows the coverage results for the new code only. In this case again, the `diff search` performs best, albeit only slightly. Exploration of the new code in this example is bottlenecked by the difficulty to cover new areas of the code.

Finally, Figure 4.5 shows how many times the symbolic exploration chose to explore constraints that were affected by the new code. This reflects the intensity with which the new code is explored. In this case, `diff search` is the clear winner, with almost twice as many affected constraints scheduled as the other heuristics.

We do not claim with these results that Oasis is better in an absolute sense. Rather, we advocate that our heuristic makes it possible to explore the new or affected code much more intensively without sacrificing coverage in general.

Interactive Symbolic Execution

Interactive symbolic execution has proven a valuable tool to implement and debug the heuristics in Oasis. Furthermore, we have used it to observe the progression of the `diff search` heuristic in the experiment described above. We report here on several observations that we made while trying to understand the results:

- Looking at the exploration of the `uServer`, it is easy to identify some areas of the code responsible for a lot of exploration. For instance, the argument processing code keeps generating

dozens of constraints for exploration. Next is the http parser. While it may be undesirable to skip those code areas altogether, using the blacklist to avoid testing them intensively should prove useful in getting to test other areas of the code more intensively.

- Any heuristic selects branches based on a set of criteria and metrics. In the case of the `diff search` heuristic, those metrics are distances measured in the control flow graph of the program. While running we observed that the relevance of the metrics varies over time. At the very beginning of the exploration, the metrics tend to be relevant and coincide well with our expectations looking at the set of constraints and their location in the source code. However, the more exploration progresses the more difficult it is to relate the distance metric to opportunities for better coverage. When this happens, a possible solution would be to switch to a different heuristic.

- When trying to guide the exploration of the code, the fine granularity of branches and constraints makes it difficult to pick a relevant set of branches manually. For instance, many branches are generated in library routines such as `strcmp()`. Grouping constraints based on the outcome of library routines would probably allow better manual control over the symbolic exploration.

While we have not yet conducted a thorough evaluation of Interactive Symbolic Execution, our preliminary experiments allowed us to gain a better understanding of the problems and difficulties faced by the search heuristic. Interactive Symbolic Execution shows promise as a key technique in advancing the state of the art in symbolic execution and automatic search heuristics.

4.5 Conclusion

In this chapter, we introduce Oasis, a new concolic execution engine. The design of Oasis is motivated by the key observation that traditional concolic and symbolic execution engines are not well suited to regression testing of software upgrades.

Oasis improves on current state-of-the-art concolic and symbolic engines by implementing a new heuristic to prioritize the exploration of new or affected code in the upgrade. Furthermore, we propose interactive symbolic execution, a new approach exposing the problem of path exploration to the tester using a graphical user interface.

The tester can use interactive symbolic execution as a learning tool to develop new heuristics or as a way to manually influence the exploration and steer it toward interesting areas of the code.

We present a preliminary evaluation of our system. Our results indicate that our heuristic successfully explores the new or affected code more intensively than existing techniques without sacrificing code coverage. We used interactive symbolic execution to understand the behavior of the heuristic and we conclude that it is a promising technique to advance the state of the art in symbolic execution.

In the future, we plan to use Oasis to find new bugs in popular open source applications. We are especially interested in using Oasis with applications that already have a large test suite to further demonstrate its ability to explore some parts of the code sooner than traditional symbolic/concolic execution engines.

```
1  Result = (inputs, constraints)
2
3  Initialization {
4    run with random input
5    put result in CACHE and CONSTRAINTS
6    run each test case
7    put result in CACHE and CONSTRAINTS
8    start worker threads
9    start main thread
10 }
11
12 Main Thread {
13   while (CONSTRAINTS is not empty) {
14     sorted = Sort(CONSTRAINTS)
15     push top entries from sorted to QUEUE
16     signal worker threads
17     wait for worker threads to finish
18     update coverage
19   }
20 }
21
22 Worker Thread {
23   while (true) {
24     if (QUEUE is empty) {
25       wait for signal from main thread
26     }
27     pop constraint C from QUEUE
28     negate C
29     input = FindSolution(!C, E)
30     {
31       run the program with input
32       put result into CACHE and CONSTRAINTS
33     }
34     signal main thread
35   }
36 }
37
38 FindSolution(C, E) {
39   query cache for input satisfying C
40   if not there, query solver
41   return input
42 }
43
44 Sort(constraints) {
45   sort constraints according to the following priority:
46   1. dangerous constraints in new code
47   2. remaining dangerous constraints
48   3. constraints at the shortest distance from
49      new uncovered code. Break ties by favoring
50      constraints affected by the new code.
51   4. constraints affected by new code
52   5. constraints at the shortest distance from
53      uncovered code
54 }
```

Listing 4.2: Search heuristic used in Oasis

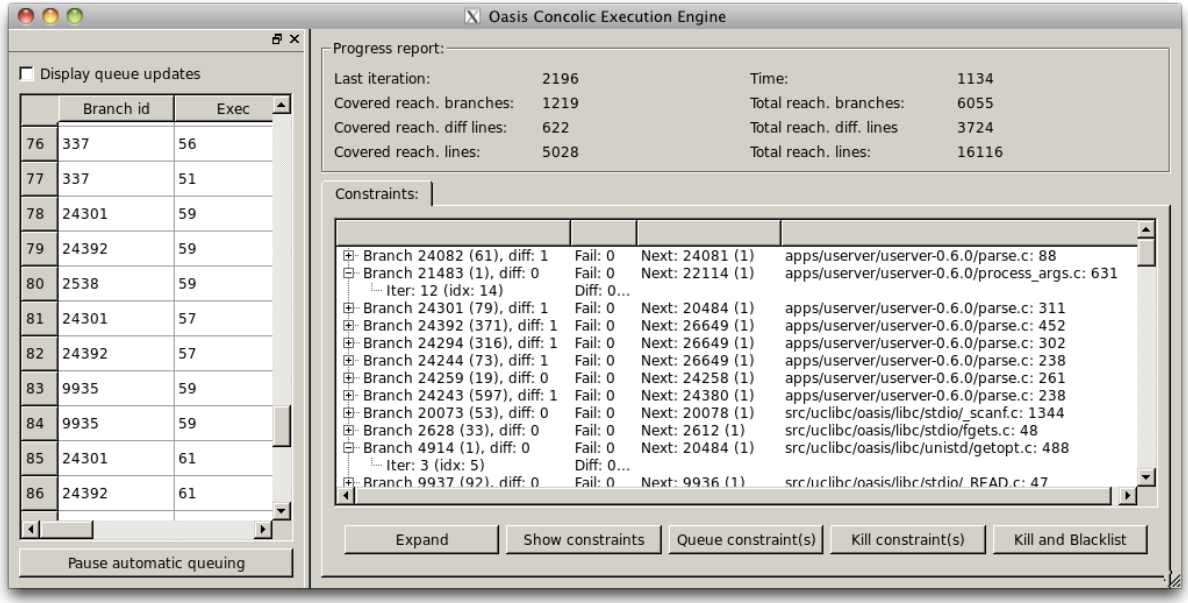

Figure 4.2: Screenshot of Oasis's graphical user interface

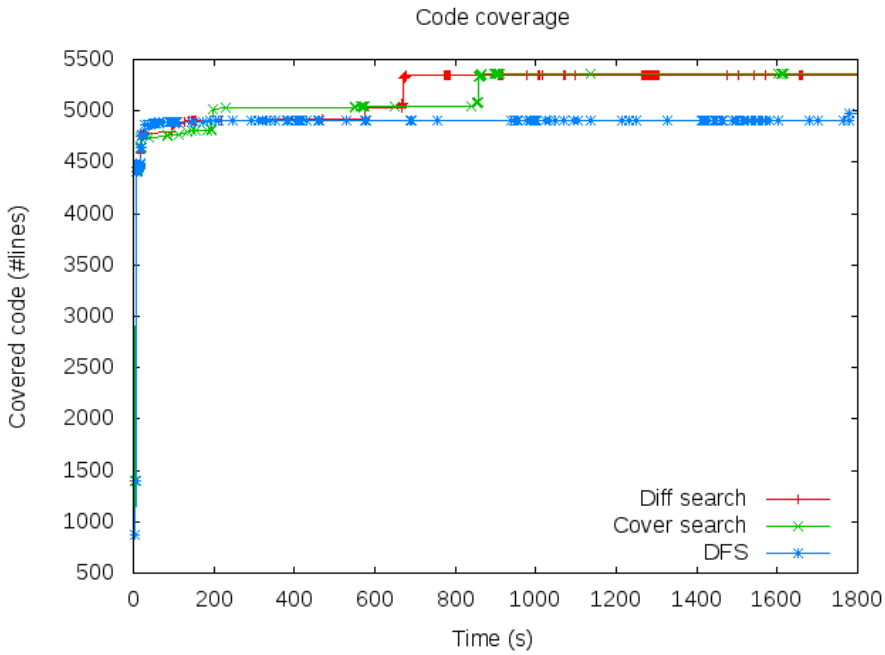

Figure 4.3: Code coverage obtained with the three configurations of Oasis on the uServer

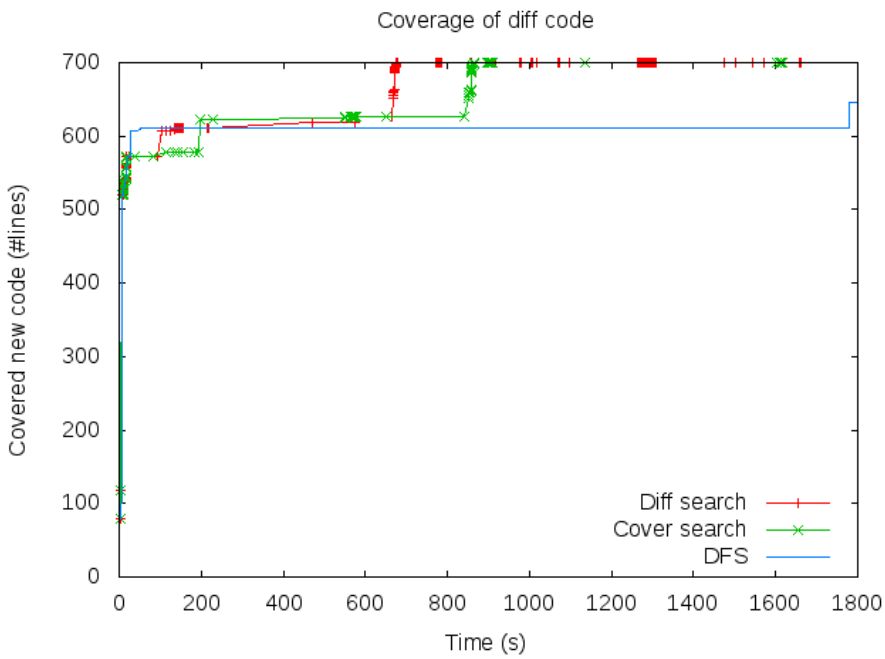

Figure 4.4: Coverage of the new code obtained with the three configurations of Oasis on the uServer

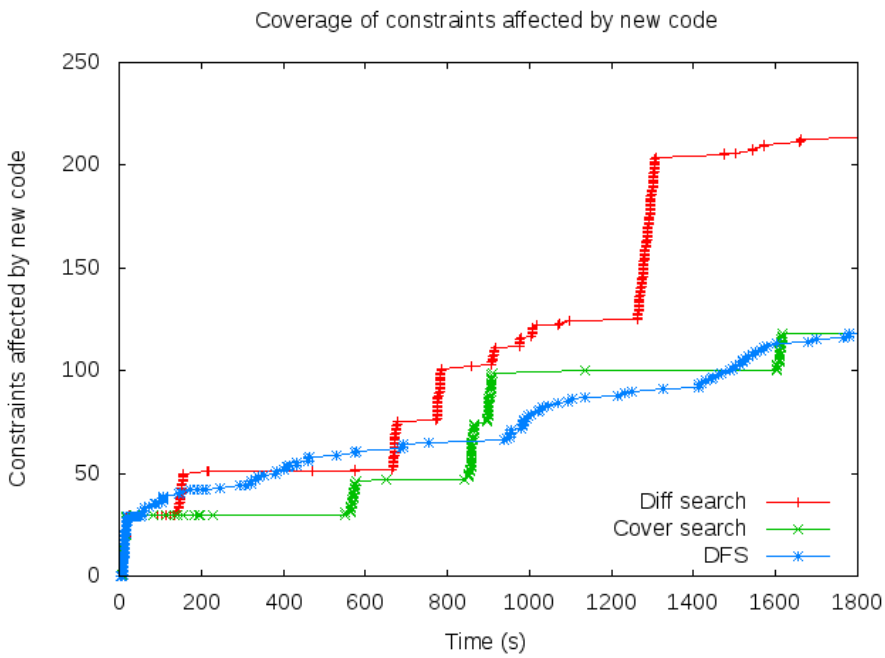

Figure 4.5: Coverage of the new or affected constraints obtained with the three configurations of Oasis on the uServer

Chapter 5

Bug Reporting

5.1 Introduction

Despite considerable advances in testing and verification, programs routinely ship with a number of undiscovered bugs. In chapter 2 we presented a survey of system administrators, detailing some of the problems facing users when deploying upgrades. In chapter 3 we describe staged deployment, a technique that significantly reduces the number of users impacted by problems by deploying upgrades in stages over clusters of users with similar environments. Subsequently, in chapter 4, we described Oasis, a state of the art concolic engine that can be used to test upgrades before they are deployed, or on user machines as part of the staged deployment protocols.

In spite of all of these efforts, some bugs are bound to remain in the software when it is deployed, and will be discovered and reported only later by the users. Debugging is an arduous task in general, and it is even harder when bugs are uncovered by users. Before the developer can start working on a fix, the problem must be reproduced. Reporting systems are meant to help with this task, but they need to strike a balance between privacy concerns, recording overhead at the user site, and time for the developer to reproduce the cause of the bug.

Reporting Systems

The current commercial state of the art is represented by the Windows Error Reporting System [24], which automatically generates a bug report when an application crashes. The bug report includes a per-thread stack trace, the values of some registers, the name of the libraries loaded, and portions of the heap. While that information is helpful, the developer must manually find the path to the bug

```
int main(int argc, char **argv) {                    1
  char option = read_option(input);                  2
  int result = 0                                     3
  if (option == 'a')                                 4
    result = fibonacci(20);                          5
  else if (option == 'b')                            6
    result = fibonacci(40);                          7
                                                     8
  printf("Result: %d\n", result).                    9
  return 0;                                          10
}                                                    11
```

Listing 5.1: A simple program computing the fibonacci sequence.

among an exponential number of possible paths. Furthermore, the information contained in the report may leak private information, which is undesirable and in certain circumstances prevents the use of the tool altogether. Recently, Zamfir and Candea have shown that symbolic execution can be used to partially automate the search for the path to the bug. However, as the manual approach, symbolic execution suffers from a potentially exponential number of paths to be explored [58].

An alternative is to record the inputs to the program at the user site. Inputs leading to failures can be transmitted to the developer, who uses them to replay the program to the occurrence of the bug. While avoiding the problem of having to search for the path to the bug, divulging user inputs is often considered unacceptable from a privacy viewpoint. Castro *et al.* generate a set of inputs that is different from the original input but still leads to the same bug [12]. Their approach does not transfer the original input to the developer, but requires input logging, whole-program replay, and invocation of a constraint solver at the user site.

A more direct approach is to record the path to the bug at the user site, for instance, by instrumenting the program to record the direction of all branches taken. In its naive form, this approach is infeasible because of the CPU, storage and transmission overhead incurred, but in this chapter we demonstrate that the approach can be optimized by instrumenting only a limited number of branches.

Our Approach

We base our work on the following three observations. First, a large number of branches do not depend on the program input, and therefore their outcome need not be logged, because it is known a priori. We denote branches whose outcome depends on program input as *symbolic*. Other branches are denoted as *concrete*.

Consider the example program in Listing 5.1. Depending on

the input parameter, the program computes the fibonacci number F_n for one of two different values of n. The only input to this program is the parameter that indicates for which value to compute the fibonacci number. While this program may have many branches (especially in the $fibonacci$ function, not shown for conciseness), it is sufficient to know the outcome of the branches at line 4 and 6 to fully determine the behavior of the program. Indeed, those two branches are the only symbolic ones, and it suffices to record their outcome.

A second, related observation is that application branches are typically either always symbolic or always concrete. In other words, it is rare that a particular branch at some point in the execution depends on input and at other points does not. An example of this can also be seen in Listing 5.1. The branches at lines 4 and 6 always depend on input, the others never do. This property is almost always true for branches in the application, and often, albeit not always, true for branches in the libraries.

Restricting our attention to branches that do depend on the input, the third observation is that it is not strictly necessary to record the outcome of all of those. When we record all such branches, the result is a unique program path, and therefore no search is required at the developer's site. When we record a subset of those branches, their outcomes no longer define a unique path but a set of possible paths, among which the developer has to search to find the path to the bug. In other words, there is a spectrum of possibilities between (1) no recording at the user site and a search at the developer site among all possible paths, and (2) complete recording of the outcome of all branches at the user site and no search at the developer site. Various points in this spectrum constitute different tradeoffs between instrumentation overhead at the user and bug reproduction time at the developer site. (We define bug reproduction as finding a set of inputs that leads the execution to the bug, or, equivalently, finding the direction of all branches taken so that they lead the execution to the bug.) It is this tradeoff that is explored further in this chapter.

In particular, we propose three approaches for deciding the set of branches to instrument. The first approach uses dynamic analysis to determine the set of branches that depend on input. This approach is constrained in its effectiveness by the limited coverage of the program that the symbolic execution engine used for dynamic analysis can achieve in a given amount of time. It tends to under-estimate the number of branches that need to be instrumented, therefore

leading to reduced instrumentation cost but increased bug reproduction time. The time that the symbolic execution engine is allowed to execute gives the developer an additional tuning knob in the tradeoff: the more time invested in symbolic execution, the better the coverage can be, and therefore the more precise the analysis. The second approach is based on static analysis of the program. Data flow analysis is used to determine the set of branches that depend on the input. This approach is limited by the precision of the static analysis, and in general tends to over-estimate the number of branches to be instrumented. Thus, this approach favors increased instrumentation cost in exchange for reduced bug reproduction time. The third approach combines the above two. It uses symbolic execution for a given amount of time, and then marks the branches that have not yet been visited according to the outcome of the static analysis.

When a bug in the program occurs, the developer runs the program in a modified symbolic execution engine that takes the partial branch trace as input. At each instrumented branch, the symbolic execution engine forces the execution to follow the branch direction specified by the log. In case a symbolic branch has not been logged, the engine explores both alternatives. Symbolic execution along the incorrect alternative eventually causes a further branch to take a direction different from the one recorded in the log. The symbolic execution engine then aborts the exploration of this alternative, and turns its attention to the other, correct branch direction.

Non-deterministic events add another dimension to the tradeoff between logging overhead and bug reproduction time. Either we log all non-deterministic events during execution so that we can reproduce them exactly during replay, but we add overhead at runtime. Or we do not log all of them, but then a non-deterministic event during replay may produce an outcome different from the one during actual execution. The importance of this tradeoff is amplified if some branches are not logged. If all branches are logged, then with high likelihood a different outcome of a non-deterministic event during replay is detected quickly, because a subsequent branch takes a direction different from the one logged during execution. If, however, not all branches are logged and in particular branches that follow the non-deterministic event are not logged, then the replay may require considerable searching to discover the path followed during the actual execution. We explore this tradeoff for system calls, one of the principal sources of non-determinism during sequential execution.

Overview of Results

We have implemented the three branch instrumentation methods described above, in addition to the naive approach that logs all branches. We explore the tradeoff between instrumentation overhead and bug reproduction time using an open-source Web server, the *diff* utility, and four coreutils programs. We find that the combined approach strikes a better balance between instrumentation overhead and bug reproduction time than the other two. Moreover, it enables bug reproduction times that are only slightly higher than those for the approach based solely on static analysis. In contrast, this latter approach only marginally reduces logging overhead compared to logging all branches, whereas the approach based solely on dynamic analysis leads to excessively long times to reproduce bugs. More concretely, our results show that the combined approach reduces instrumentation overhead between 10% and 92%, compared to the approach based solely on static analysis. At the same time, it always reproduces the bugs we considered within the allotted time (1 hour), whereas the dynamic approach fails to do so in 6 out of 16 cases. In all circumstances, we find that selectively logging the results of system calls is advantageous: it limits bug reproduction time, and adds only marginally to instrumentation overhead.

Contributions

The contributions of this chapter are:

1. The use of symbolic execution prior to shipping the program to discover which branches depend on input and which not.

2. An optimized approach to symbolic execution for bug reproduction that is guided by a symbolic branch log collected at the user's site.

3. The exploration of the tradeoffs between the amount of predeployment symbolic execution, the instrumentation overhead resulting from logging the outcome of branches, and the time necessary to reproduce a bug at the developer's site.

4. A combined static-dynamic method for deciding which branches to instrument. The method leads to a better tradeoff than previous systems, making the approach of logging branches more practical and reducing the debugging time.

5. A quantification of the impact of logging the result of selected system calls, demonstrating that it only marginally increases

the instrumentation overhead, but considerably shortens the bug reproduction time.

Roadmap

The rest of this chapter is organized as follows. Section 5.2 describes the program analysis and instrumentation methods we study. Section 5.3 shows how we modify a symbolic execution engine to take as input a partial branch recorded at the user site to reproduce a bug. Section 5.4 describes some implementation details and our experimental methodology. Section 5.5 presents the results for an open-source Web server, diff, four coreutils programs, and two microbenchmarks. Section 5.6 discusses the results and our future work. Finally, Section 5.7 concludes the chapter.

5.2 Program Analysis and Instrumentation

Our approach involves analyzing the program to find the symbolic branches and instrument them for logging. We study both dynamic and static analyses. Our instrumentation may use the results of the dynamic analysis only, those of the static analysis only, or combine the results of both analyses. Next, we describe our analyses and instrumentation methods.

Dynamic Analysis

Our dynamic analysis is based on symbolic execution. We mark input to the program as symbolic and then use symbolic execution to determine whether or not a branch condition depends on input.

Symbolic execution repeatedly and systematically explores all paths (and thus branches) in a program. In the particular form of symbolic execution used in this dissertation (sometimes called concolic execution [51], see chapter 4), the symbolic engine executes a number of runs of the program, each with a different concrete input. Initially, it starts with a randomly chosen set of values for the input and proceeds down the execution path of the program. At each symbolic branch, it computes the branch condition for the direction of the branch followed, and adds this condition to the constraint set for this run. When the run terminates, the overall constraint set describes the set of branch conditions that have to be true for the program to follow the path that occurred in this particular run. One of the conditions is then negated, the constraint set is solved to obtain a new input, and a new run is started.

We initially mark *argv* as symbolic, as well as the return values of any functions that return input. During symbolic execution, we track which variables depend on input variables, and mark those as symbolic as well. When we execute a branch for the first time, we label it symbolic if any of the variables on which the branch condition depends is symbolic, and concrete otherwise. If during further symbolic execution we revisit a branch labeled symbolic, it stays that way. If, however, we revisit a branch labeled concrete, and now its branch condition depends on at least one symbolic variable, we relabel that branch as symbolic.

Symbolic execution tries to explore all program paths, and is therefore very time-consuming. If it is able to explore all paths, then all branches are visited, with some marked symbolic and some marked concrete. However, this usually can only be done for very small programs. For others, it is necessary to cut off symbolic execution after some amount of time. As a result, at the end of the analysis, in addition to branches labeled symbolic and concrete, some branches remain unlabeled.

All branches marked symbolic are indeed symbolic, but some branches marked concrete may actually be symbolic, because symbolic execution was terminated before the branch was visited with a condition depending on input. The unvisited branches may be either symbolic or concrete.

Static Analysis

We use interprocedural, path-sensitive static analysis, in which we use a combination of dataflow and points-to analysis. The basic idea of the algorithm is to identify the sources of input (typically I/O functions or arguments to the program), and construct a list of variables whose values depend on input and are thus symbolic. Symbolic branches are then identified as the branches whose condition depends on at least one symbolic variable.

The algorithm works by maintaining a queue of functions to analyze. Initially, the queue only contains the *main* function. New entries are queued in as function calls are discovered. The set of symbolic variables is initialized to *argv*. In the initialization, the functions that are normally used to read input are marked as returning symbolic values.

Algorithms 1 and 2 show a simplified version of the algorithm that analyzes each function. Each instruction in the program is visited, and the $doInst$ method is called. The dataflow algorithm takes care of loops by using a fixed-point algorithm so that instructions

Algorithm 1: Static analysis algorithm propagating symbolic information (simplified)

```
   /* called on each instruction in the program as many
      times as required by the fixed point dataflow
      algorithm                                          */
1  method doInstr(instruction i):;
2  begin
3      match i with begin
           /* assignment                                 */
4          case target_variable = expression;
5          begin
               /* If any of the variables referenced by
                  expression symbolic is symbolic mark
                  targe_variable symbolic                */
6              if isSymbolic(e) then
7                  makeSymbolic(target_variable);
8              end
9          end
           /* function call                              */
10         case: target_variable = fun_name(parameters);
11         begin
12             symbolic_params =
               getSymbolicParameters(parameters);
               /* If we already visited fun_name with this
                  combination of symbolic parameters,
                  propagate symbolic flag                */
13             if alreadyVisited(fun_name, symbolic_params) then
14                 if returnsSymbolicMemory(fun_name) then
15                     makeSymbolic(target_variable);
16                 end
17             end
18             else
                   /* fun_name not visited yet.  Queue it and
                      stop analysis of current function.  The
                      algorithm will return to this location
                      once fun_name has been visited     */
19                 queueFunction(fun_name, symbolic_parameters,
                   i);
20                 return abort
21             end
22         end
23     end
24     return continue
25 end
```

in a loop body are revisited (and *doInst* is called) only as long as the algorithm output changes.

For an assignment instruction (i.e., an instruction of the form *variable* = *expression*;), *doInstr* resolves the variables to which the expression may be pointing, and checks whether any of these is already known to be symbolic. If this is the case, it adds *variable* to the list of symbolic variables, otherwise it continues. If the in-

Algorithm 2: Static analysis algorithm identifying symbolic branches (simplified)

```
   /* called on each control flow statement of the program.
      */
 1 method doStatement(statement s): begin
 2 |   match s with;
 3 |   begin
 4 |   |   Branch(condition_expression)  begin
 5 |   |   |   if isSymbolic(condition_expression) then
 6 |   |   |   |   logThisBranch();
 7 |   |   |   end
 8 |   |   end
 9 |   end
10 |   return continue;
11 end
```

struction is a function call, the algorithm first checks whether the function has already been analyzed or not. If it has, it looks up the results to determine whether with the current set of parameters the function can propagate symbolic memory or not, and updates the list of symbolic variables accordingly. If the function has not yet been visited [1], the algorithm enqueues it for analysis with a reference to the current instruction, so that analysis of the current function can resume when the function has been visited.

The *doStatment* method in algorithm 2 is called by the dataflow framework on each control flow statement. For if statements, it resolves the list of variables to which the condition expression may be pointing. If any of them is symbolic, the branch is labeled symbolic.

While the algorithm in Figure 1 is simplified for the sake of clarity, our actual implementation handles the fact (1) that symbolic variables can be propagated to global variables; (2) that the state of global variables changes depending on the path that is being analyzed; and (3) that functions may propagate symbolic variables not only to their return variables, but also to their parameters (passed by reference) or to global variables.

Static analysis is imprecise because the points-to analysis tends to over-estimate the set of aliases to which a variable may point. As a result, all symbolic branches in the program are labeled symbolic by the static analysis, but some concrete branches may also be labeled symbolic. All branches labeled concrete are indeed concrete.

[1] More precisely, if the function has not been visited with the particular combination of symbolic and concrete parameters encountered here.

Program Instrumentation

The developer instruments the branches in the program before the code is shipped. We consider four methods for instrumentation:

- *dynamic* instruments branches according to dynamic analysis.
- *static* instruments branches according to static analysis.
- *dynamic+static* instruments branches according to a combination of dynamic and static analysis.
- *all branches* instruments all branches.

Regardless of which method is used, the list of instrumented branches is retained by the developer, because it is needed to reproduce the bug (Section 5.3).

Dynamic method. After dynamic analysis, branches are labeled symbolic or concrete, or remain unlabeled. The *dynamic* method only instruments the branches labeled as symbolic. By the nature of the dynamic analysis, we are certain that these branches are symbolic. We do not instrument the branches labeled as concrete, since application branches are typically either always symbolic or always concrete. We also do not instrument the unlabeled branches. The *dynamic* method thus potentially underestimates the number of branches that need to be instrumented. In essence, *dynamic* favors reducing instrumentation overhead at the expense of increased bug reproduction time.

Static method. After static analysis, branches are labeled symbolic or concrete. We instrument the branches labeled as symbolic. By the nature of static analysis, the *static* method guarantees that all symbolic branches are instrumented, but it may instrument a number of concrete branches as well. *Static* therefore favors bug reproduction time at the expense of increased instrumentation overhead.

Dynamic+static method. In the combined method, we run both the static and the dynamic analysis, the latter for a limited time. The dynamic analysis labels branches as symbolic or concrete, or they may remain unlabeled. The static analysis labels branches as symbolic or concrete. The combined method instruments the branches (1) that are labeled symbolic by the dynamic analysis, and (2) that are labeled symbolic by the static analysis, with the exception of those labeled concrete by the dynamic analysis. In other words, when a branch is not visited by dynamic analysis, we

instrument it based on the outcome of the static analysis, because this is the only information about this branch. When a branch is visited by dynamic analysis, we instrument it based on the outcome of this analysis. For branches labeled symbolic by dynamic analysis, this is obvious as they are guaranteed to be symbolic and have been labeled symbolic by static analysis as well. For branches labeled concrete by dynamic analysis, this means that we potentially override the outcome of static analysis which may have labeled these branches symbolic. The reasons for this decision are that (1) static analysis may conservatively label concrete branches symbolic, due to an imprecise points-to analysis, and (2) application branches are typically always concrete or always symbolic, as mentioned above.

Dynamic+static may be imprecise in two ways. Symbolic branches may or may not be instrumented. The latter case occurs if they are left concrete by the symbolic execution (for instance, due to the limited coverage of the symbolic execution). Concrete branches may or may not be instrumented. The former case occurs when the static analysis mistakenly labels them as symbolic and they are not visited during symbolic execution.

Although seemingly suffering from a greater degree of imprecision than the other methods, our evaluation shows that this method actually leads to the best tradeoff between instrumentation overhead and time necessary to reproduce the bug.

Logging system calls. In addition to deciding which branches to instrument, we also consider the choice of whether or not to log the results of certain system calls. Doing so adds to the runtime overhead, but can be very beneficial for system calls that can produce a large number of possible outcomes during replay. For example, consider a *select()* system call for reading from any of N file descriptors. Without information about which descriptors became ready and when, symbolic execution during replay would have to explore all combinations of N available descriptors upon each return from *select()*. To avoid having to explore all possible combinations, we instrument the code to log the descriptors that are available when a call to *select()* returns. During replay, we simply re-create these conditions. For the same reasons, it makes sense to instrument calls to *read* to log the number of bytes read.

We log the results of all system calls for which logging considerably simplifies replay, including *select()* and *read()*. The input data itself is never logged. In principle, all instrumentation methods can be combined with logging system calls.

5.3 Reproducing a Bug

Replay Algorithm

We use Oasis (see chapter 4 to reproduce bugs. The following information is available to the engine prior to bug reproduction: a list of all instrumented branches (saved when the program was instrumented – see Section 11), and the bitvector indicating which way the instrumented branches were taken (one bit for each instrumented branch taken during execution at the user site). When the symbolic execution engine encounters a branch, it immediately knows whether or not the branch is symbolic, because it can check whether the branch condition depends on the input.

We refer to a run of the symbolic execution engine as the execution of the engine with a single set of inputs, until it either finds the path to the bug or aborts. A run is aborted when the engine discovers that it is on a path that deviates from the path described by the received bitvector. Each run is started with the bitvector as received from the user site. A constraint set is associated with each run, describing the path followed by the run through the program, and consisting of the conjunction of the conditions for the branch directions taken so far in the run. To later explore alternative paths should the current run abort, the engine also maintains a list of pending constraint sets, describing these alternative unexplored paths.

The engine performs a number of these runs. The initial run is done with random inputs. Subsequent runs use an input resulting from the solution of a constraint set by the constraint solver. For example, in Listing 5.1, the set of constraints: {*not (option == 'a'); option == 'b'*} needs to be solved to take the program in the `else if` branch at line 6. When visiting the `else if` branch at line 6, the engine puts in the pending list the following constraint set: {*not (option == 'a'); not (option == 'b')*}.

During a run, the engine proceeds normally for instructions other than branches. For branches, because of the imprecision of the decision of which branches to instrument, the following four cases have to be distinguished.

1. **The branch is symbolic and not instrumented.** The constraint for the particular direction of the branch taken is added to the constraint set for the run, and symbolic execution proceeds. In addition, a new constraint set is formed by adding the negated constraint to the constraint set for the run. The

new constraint set is put on the list of pending constraint sets. The bitvector is left untouched.

2. **The branch is symbolic and instrumented:** The engine takes the next bit out of the bitvector, and compares the direction that was taken during recorded execution to the direction the symbolic execution would take with its input.

   a) If the two are the same, the constraint is added to the constraint set for the run, and the symbolic execution proceeds.

   b) If not, the constraint implied by the direction of the branch taken during recorded execution is negated and added to the constraint set for the run. This constraint set is added to the list of pending constraint sets, and the run aborts.

3. **The branch is concrete and instrumented:** The engine takes the next bit out of the bitvector, and compares if the direction that was taken during recorded execution is the same as the direction the symbolic execution would take with the current input.

   a) If yes, it proceeds further.

   b) If not, this run of the symbolic execution is aborted.

   The latter case can only occur as a result of the run earlier having taken the wrong direction on a branch that (because of insufficient instrumentation) was not instrumented but should have been.

4. **The branch is concrete and not instrumented:** The engine proceeds. The bitvector is left untouched.

When a run is aborted, the engine looks at the pending list of constraint sets, picks one, solves it, and starts another run with the input resulting from the solver. When this new run passes the branch at which it was produced, the new input, by construction, causes the symbolic execution to take the direction opposite from the one followed in the run during which the constraint set was produced.

Replay Under Different Instrumentation Methods

The *all branches* instrumentation method instruments all symbolic branches (and all concrete ones as well). Thus, cases 1 and 4 above

cannot occur with this method. Furthermore, case 3(b) cannot occur either. The reason is that the engine always proceeds past a symbolic branch in the same direction as recorded during execution. When the run hits a concrete branch, since this branch does not depend on input, the engine is bound to follow the correct direction.

The *static* method also instruments all symbolic branches, since imprecision in the points-to analysis is resolved conservatively (if a pointer might depend on input, it is flagged symbolic). Under this method, cases 1 and 3(b) cannot occur. The latter case cannot occur for the same reason above.

The *dynamic* method may not instrument all symbolic branches, because it may not run long enough to find them. Similarly, the *dynamic+static* method may fail to instrument all symbolic branches, but only when symbolic execution does not run long enough *and* static analysis is inaccurate. In these cases, a run can take the wrong direction at a symbolic branch that was not instrumented, and later hit a concrete branch for which the input fails to satisfy the branch condition. In this case, the engine needs to back up and explore an alternative direction at a symbolic but uninstrumented branch. The constraint sets for these alternative directions reside on the pending list. The search can use any heuristic for deciding which constraint set to pick from the pending list. We currently use a simple depth-first approach.

Replaying System Calls

As mentioned in Section 11, we consider scenarios with and without instrumentation for logging the results of certain system calls.

When these system calls are not logged, we replay them using models of their behavior. The models use symbolic variables to allow the engine to reproduce any behavior that may be produced by the kernel. For instance, for the *read()* system call, we use a symbolic variable for the return value that determines how much input is read. This symbolic variable is constrained to be between -1 and the amount of input requested. During replay, a program executing the *read()* call initially returns the amount of (symbolic) input requested, and execution carries on. Because the return value of the system call is symbolic, if the program checks it in a branch and if the branch has been logged, the log specifies which direction needs to be taken. If that direction fails, the symbolic execution engine aborts the run. Eventually, the number of bytes actually read at the user's site is found.

When these system calls are logged, we replay their execution based on the logs. During replay, the calls for which there are logged results always return exactly the recorded value. Thus, the symbolic execution engine need not search for the actual call results.

5.4 Implementation and Methodology

Software. For program instrumentation and analysis, we use CIL (C Intermediate Language [42]), which is a collection of tools that allow simple analysis and modification of C code.

For static analysis we start by merging all the source code files of the program in one large C file. This allows us to run the analysis on the whole program, making the results more accurate. We then use two CIL plug-ins for the dataflow and points-to analysis.

For symbolic execution, we use Oasis, our home-grown concolic execution engine for C programs described in chapter 4. The engine instruments the C program and links it with a runtime library for logging constraints.

We also use CIL to instrument the branches in the program. The instrumentation simply uses a bit per branch in a large buffer, and flushes the buffer to disk when it is full. We use a buffer of 4KB in order to avoid writing to disk too often. We do not use any form of online compression, as this would impose additional CPU overhead. Moreover, we could have used a simple branch prediction algorithm to avoid logging all instrumented branches, but this would have required recording the program location for each logged branch. This approach would have required at least another 32 bits of storage per branch, probably ruining any savings obtained by the prediction algorithm.

Benchmarks. We first evaluate the instrumentation overhead in isolation using two microbenchmarks. Next, we reproduce real bugs previously reported in the coreutils programs [10]. Then, we evaluate the tradeoff between instrumentation overhead and bug reproduction time using an open-source Web server, the *uServer* [45]. The *uServer* was designed for conducting performance experiments related to Internet service and operating system design and implementation. We use version 0.6.0, which has approximately 32K lines of code. In our final experiment, we again evaluate the tradeoff between instrumentation overhead and bug reproduction, but this time with the *diff* utility. *Diff* is an input-intensive application that pinpoints the differences between two files provided as input. It

contains about 22K lines of code. For all experiments we link the programs with the uClibc library [55].

We study five configurations of each benchmark: four resulting from our four instrumentation methods plus a configuration called *none*, which involves no instrumentation. For the instrumented benchmarks, unless mentioned otherwise, selective system call logging is turned on.

Hardware. Our experimental setup consists of two machines with two quad-core 2.27-GHz Xeon CPUs, 24 GBytes of memory, and a 160-GByte 7200rpm hard disk drive.

5.5 Evaluation

Microbenchmarks

We evaluate the cost of the instrumentation by using two simple microbenchmarks. The first microbenchmark comprises a loop that increments a counter 1 billion times. The loop condition (checking the loop bound) consists of a single branch, executing as many times as there are iterations.

We compare the *none* (no instrumentation) and *all branches* versions of this microbenchmark using the Linux *perf* profiler. The results show that the branch logging instrumentation takes 17 instructions on average, including the periodic flushing of the log to disk. In terms of running time, we see an average cost of 3 nanoseconds per instrumented branch (with an average of 2.1 instructions per cycle), for a total overhead of 107%. While considerable, this overhead is still lower than that reported in ODR [1] (about 200% just for the branch recording). The most likely reason is that our instrumentation only logs one bit per branch, while ODR uses a more complicated branch prediction algorithm.

We run a second microbenchmark consisting of the example in Listing 5.1, which computes the fibonacci sequence for one of two numbers. We instrument this program using the four configurations of our system. Not surprisingly, the configurations other than *all branches* instrument only the symbolic branches corresponding to lines 4 and 6. The results are consistent with our previous microbenchmark: an average overhead of 17 instructions per instrumented branch or about 3 nanoseconds. The *all branches* configuration suffers from a total overhead of 110%, whereas the three others do not incur any noticeable overhead because only two branches are logged.

Coreutils

We now evaluate our approaches using four real bugs in different programs from the Unix coreutils set: *mkdir*, *mknod*, *mkfifo*, and *paste*. We ran the programs with up to 10 arguments, each 100 bytes long.

Branch behavior. Recall that our approach is based on two assumptions about branches: (1) that there are many (concrete) branches whose outcomes do not depend on input, and (2) that if a branch is executed once with a concrete (symbolic) branch condition, it is most likely always executed with a concrete (symbolic) condition.

To check these assumptions, we modify our symbolic execution engine to run with concrete inputs (instead of generating concrete inputs itself to explore different paths). In addition, at each executed branch, we record whether it is executed with a symbolic or a concrete branch condition.

We show the results of a sample run of *mkdir* in Figure 5.1; the graphs for the other 3 programs are similar. The figure shows per-branch-location statistics for this experiment. We use the term "branch location" to mean the location of a branch in the source code, and "branch execution" to mean the actual execution of a branch location during run time. Each point on the x axis denotes a branch location that is executed at least once. The y axis shows how many times each branch is executed. The gray bars denote the overall number of branch executions, whereas the black bars denote the number of executions with a symbolic condition. The black bars are therefore a subset of the gray bars.

As the figure illustrates, only a limited number of branch locations is responsible for all the symbolic branch executions. Furthermore, where black bars are present, they completely cover the gray bars. This shows that a particular branch is executed either always with concrete conditions or always with symbolic conditions. These observations support our two assumptions.

Instrumentation overhead. Figure 5.2 shows the CPU time associated with the instrumentation for *mkdir* (again, the results for the other programs are similar). Those results are obtained by running the program in the symbolic execution engine for one hour. The figure shows that the time is almost identical for the *dynamic*, *dynamic+static*, and *static* configurations. The static and dynamic analyses produce accurate results in those programs. The *all branches* configuration is the slowest, with an overhead of 31%.

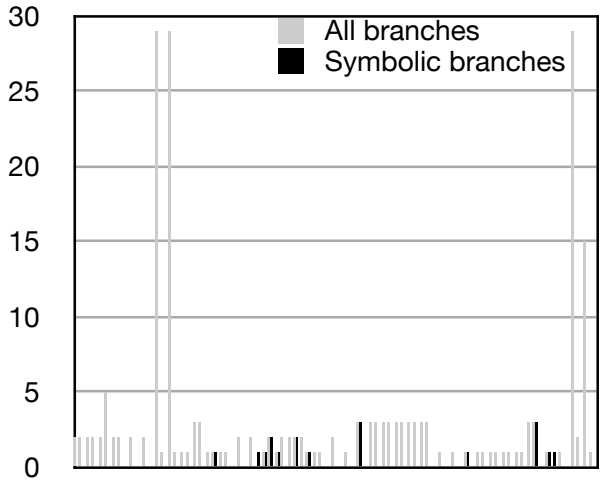

Figure 5.1: Number of executions of each branch in a sample run of *mkdir*. The overlaid black bars represent the branches executed with symbolic conditions.

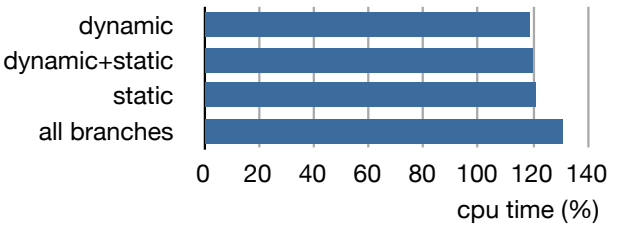

Figure 5.2: CPU time of *mkdir* instrumented with the four configurations of our system. Results are normalized to the non-instrumented version.

Reproducing bugs. Each program suffers a crash bug that only manifest itself when a very specific combination of arguments is used. For instance, the bug in *paste* occurs with the following command line:

```
paste -d\\ abcdefghijklmnopqrstuvwxyz
```

Table 5.1 shows the time needed to reproduce the crash bugs in the four programs. The programs being relatively small, symbolic execution is able to cover most of the important branches in a very short amount of time, and static analysis produces accurate results. Thus, we can reproduce the bug in less than two seconds in all of the four instrumented configurations.

These bugs have also been used to evaluate ESD [58]. Interestingly, ESD took significantly more time to reproduce the bugs (between 10 and 15 seconds, albeit with no runtime overhead). This can be attributed to the fact that our system essentially knows the exact

Program	Replay time
mkdir	1 sec
mknod	1 sec
mkfifo	1 sec
paste	1.5 sec

Table 5.1: Time needed to replay a real bug in four coreutils programs. The results are the same with all four configurations of our system.

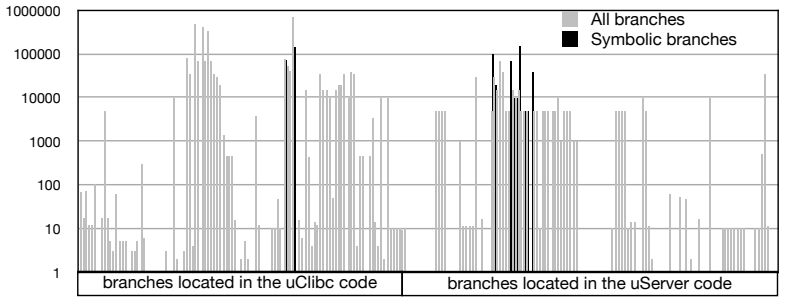

Figure 5.3: Number of executions of each branch location in a sample run of the *uServer*. The y axis is in log scale.

path to the bug (during bug reproduction), whereas ESD needs to search many paths.

uServer

We further evaluate our system using a much larger application (32K lines of code), the *uServer* [45], an open-source Web server sometimes used for performance studies. Unlike the coreutils programs, which are very small, this benchmark elucidates the differences between the approaches for deciding which branches to instrument, and the tradeoff between instrumentation overhead and bug reproduction time.

Branch behavior. We run the *uServer* in our modified symbolic execution environment with 5,000 HTTP requests to demonstrate the nature of the branches (symbolic or concrete). In total, approximately 18 million branches are executed, out of which only 1.8 million or roughly 10% are symbolic. These 1.8 million symbolic branches correspond to multiple executions of the same 53 branch locations in the program.

Figure 5.3 shows per-branch-location statistics for this experiment. As we can see, most of the black bars entirely cover their corresponding gray bars. This means that these particular branch

Version	**# of instrumented branch locations**	
	LC	HC
dynamic	78	246
dynamic + static	1654	1490
static	2104	
all branches	5104	

Table 5.2: Number of branch locations instrumented in the *uServer* with the different configurations of our system.

locations are executed either always with concrete conditions, or always with symbolic conditions. However, the situation is slightly different for the branches in uClibc, where in some cases the black bars almost but not completely cover the gray bars. This situation corresponds to library functions that are sometimes called with concrete values. In this experiment, the number of those cases was very small.

The figure also shows that most branches are executed in the library (81%). However, only 28% of the symbolic branches are executed in the library.

Identifying symbolic branch locations. In total, there are 5104 branch locations in the *uServer* code (and 8516 in uClibc).

Table 5.2 shows the number of instrumented branches in the *uServer* for each configuration of our system. We symbolically execute the *uServer* using 200 bytes of symbolic memory for each accepted connection, and for each file descriptor. We stop the symbolic execution phase after one hour and two hours, obtaining a coverage of 20% (denoted LC for lower coverage) and 33% (HC for higher coverage), respectively. Running longer does not significantly improve branch coverage.

Out of a total of 5104 branches, *static* marks 2104 as symbolic, *dynamic* 246, and *dynamic+static* 1490, in the HC configuration. In addition, with *dynamic*, 1434 branches are marked as concrete, and the remaining branches are not visited.

For the static analysis tool, we need to merge all the source files of both the application and the library. Unfortunately, doing so resulted in a file too large for the points-to analysis to handle.[2] Therefore we perform static analysis only on the *uServer* application code. All branches in the library are treated as symbolic by the static analysis.

[2] After six hours, the analysis had made little progress and we aborted it.

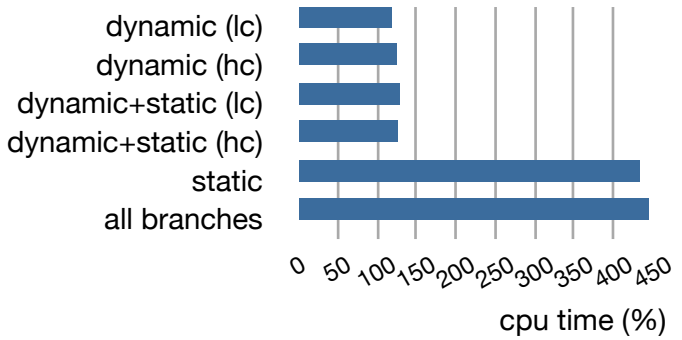

Figure 5.4: CPU time of the *uServer* instrumented with the four configurations of our system.

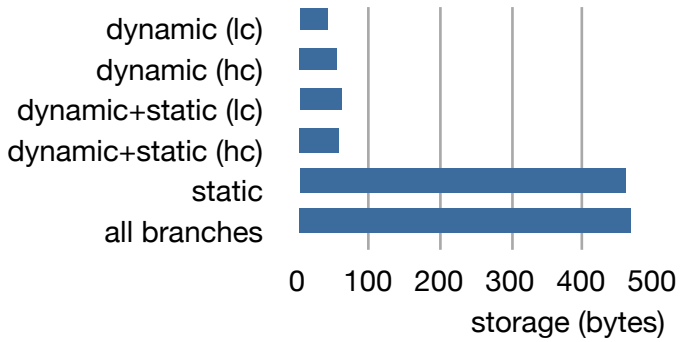

Figure 5.5: CPU time (a) and storage requirement (b) of the *uServer* instrumented with the four configurations of our system.

With less coverage, there are more branches left unvisited, and therefore more opportunity for the static phase to mark them symbolic. This is why there are fewer instrumented branches in the *dynamic* configuration and more in the *dynamic+static* configuration. Those numbers are used to highlight the effect of branch coverage in our approach.

Instrumentation overhead. We use the httperf [36] benchmarking tool to compare the performance of the configurations for the *uServer*. We run httperf with a static workload saturating the CPU core on which the *uServer* is running. We consider three performance metrics: number of instrumented branches executed, CPU time, and storage requirement of the instrumentation. All three are roughly proportional to each other.

Figure 5.4 shows CPU time (relative to the non-instrumented version). As we can see, the overhead of *all branches* is significant. The results of *static* are only marginally better, since it instruments all branches in the uClibc library.

The two configurations using dynamic analysis perform notably better. The overhead is 17% and 20%, respectively, for the *dynamic* and *dynamic+static* configurations. This is not surprising for *dynamic*, as it instruments only 284 branch locations. The *dynamic+static* configuration instruments many more branch locations, but still far fewer than *all* or *static*.

The coverage obtained with symbolic execution affects the instrumentation overhead of *dynamic* and *dynamic+static*. Increased coverage increases the overhead of *dynamic*, since symbolic execution instruments more branches. In contrast, increased coverage leads to reduced instrumentation in *dynamic+static*, corresponding to branches marked symbolic by the static analysis, left unvisited by the dynamic analysis with low coverage, and marked concrete by the dynamic analysis with high coverage.

Figure 5.5 shows the storage requirements per HTTP request processed by the Web server. The storage overhead is reasonable; around 50 bytes per request in the *dynamic* and *dynamic+static* configurations. This is roughly the same number of bytes in a typical entry in the access log of the Web server.

Both the processing and storage overheads are more significant in the *static* and *all branches* configurations. These configurations represent worst-case scenarios for the areas of the code that are not covered by the symbolic execution.

When a bug occurs, the branch log has to be transferred to the developer. Compression can be used to reduce the transfer time. We observe a compression ratio of 10-20x using gzip. In Section 5.6, we discuss how to deal with long-running executions, which could generate extremely large branch logs.

Reproducing bugs. To evaluate the amount of effort required to reproduce a bug, we run the *uServer* with five different input scenarios. To demonstrate the impact of code coverage on our approach, we design those scenarios to hit different code areas of the HTTP parser. More specifically, we use HTTP queries of various lengths (between 5 to 400 bytes), with different HTTP methods (e.g., GET, POST) and parameters (e.g., Cookies, Content-Length). We crash the server by sending it a SEGFAULT signal after sending it the input, making sure it crashes at the same location in the code for all four versions. After replay, we verify that each configuration produced input that correctly leads to the same location.

Table 5.3 shows the bug reproduction times for all four instrumented versions of the five scenarios. Table 5.4 shows the corresponding number of symbolic branch locations logged and not

Version	**Exp. 1**		**Exp. 2**		**Exp. 3**		**Exp. 4**		**Exp. 5**	
	LC	HC	LC	HC	LC	HC	LC	HC	LC	HC
dynamic	27s	27s	2877s	79s	∞	170s	∞	287s	∞	168s
dynamic+static	27s	27s	79s	79s	532s	170s	175s	175s	248s	168s
static	27s		79s		170s		175s		168s	
all branches	27s		79s		170s		175s		168s	

Table 5.3: Time in seconds needed to reproduce each of the five input scenarios to the *uServer* with the four instrumented configurations of our system. The infinity symbol means that the experiment did not terminate in one hour.

Version		# of symbolic branch locations logged / corresponding # of executions		# of symbolic branch locations NOT logged / corresponding # of executions	
		LC	HC	LC	HC
Exp. 1	dynamic	18 / 112	18 / 112	0	0
	dynamic+static	18 / 112	18 / 112	0	0
	static	18 / 112		0	
	all branches	18 / 112		0	
Exp. 2	dynamic	25 / 129913	39 / 2215	11 / 23062	0
	dynamic+static	36 / 2105	39 / 2215	3 / 110	0
	static	39 / 2215		0	
	all branches	39 / 2215		0	
Exp. 3	dynamic	25 / 554617	42 / 28848	17 / 48485	0
	dynamic+static	45 / 10971	42 / 28848	3 / 1023	0
	static	42 / 28848		0	
	all branches	42 / 28848		0	
Exp. 4	dynamic	24 / 236608	43 / 24012	21 / 185945	6 / 268
	dynamic+static	46 / 11089	48 / 12111	3 / 1023	1 / 1
	static	49 / 12112		0	
	all branches	49 / 12112		0	
Exp. 5	dynamic	25 / 410723	44 / 29136	15 / 45706	0
	dynamic+static	46 / 54539	44 / 28785	3 / 3391	0
	static	44 / 28785		0	
	all branches	44 / 28785		0	

Table 5.4: Number of symbolic branch locations and symbolic branch executions logged and not logged for each configuration of each experiment of Table 5.3.

logged, as well as the number of actual symbolic branch executions. Both tables include the configuration with low coverage (LC) and high coverage (HC).

Unsurprisingly, the *all branches* and *static* versions, which instrument all symbolic branches, perform best. Of course, these versions do so at high runtime and storage overheads (Figures 5.4 and 5.5).

Dynamic+static in most cases performs only slightly worse than *static*, despite the much lower instrumentation overhead of the former configuration. *Dynamic* comes last, with many LC experiments not finishing in one hour. This is not surprising, as the number of branches identified as symbolic, and therefore the amount of logging, is very low. In fact, Tables 5.3 and 5.4 show that the number of symbolic branch locations not logged is well correlated with the replay time. As soon as replay encounters more than a dozen symbolic branch locations that are not instrumented, the replay time exceeds one hour. An approach that does not instrument the code at all, would result in even longer bug reproduction times.

Dynamic+static obtains similar results regardless of coverage. The reason is that symbolic execution may incorrectly classify some branches as concrete, and thus slow down the search. When running longer, those branches may later be marked symbolic, therefore correcting the error. In our experiment, those differences have a marginal effect on *dynamic+static*, which again suggests that this does not happen frequently.

Impact of logging system calls. By default, we log the results for some key system calls (Section 11). For instance, we log the return value of the *read()* system call and the order of ready file descriptors from *select()* calls.

The measurements in Figures 5.4 and 5.5 include the overhead of logging these return values. As we only log a limited number of values for a few system calls, logging these values introduces little extra work compared to the logging of the branches. As a result, when not logging system call results, the overhead is reduced by a marginal 0.2%.

Tables 5.5 and 5.8 present the bug reproduction times of two of our experiments without logging any system calls (we omit the three other experiments for brevity; their results are similar). All configurations of our system take significantly longer to replay, as the symbolic execution engine needs to determine the exact return values of the selected system calls. The *dynamic* and *dynamic+static* configurations are further penalized, since the back-tracking needed by the unlogged symbolic branches compounds the search for the re-

Version	Exp. 1		Exp. 4	
	LC	HC	LC	HC
dynamic	112s	112s	∞	712s
dynamic+static	112s	112s	991s	694s
static	87s		362s	
all branches	56s		343s	

Table 5.5: Time in seconds needed to reproduce two input scenarios with the *uServer* when not logging system call results. The infinity symbol means that the experiment did not terminate in one hour.

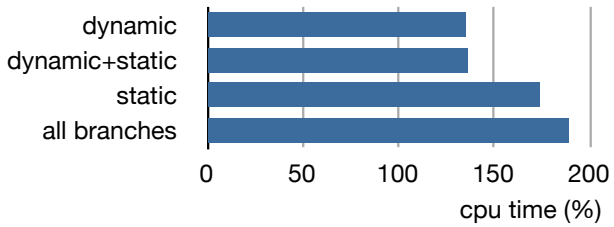

Figure 5.6: CPU time of *diff* instrumented with the four configurations of our system. Results are normalized to the non-instrumented version.

turn values. Interestingly, the *static* configuration performs slightly slower than *all branches*, whereas when logging system calls it performs identically. The reason is that fewer concrete branches are logged, therefore the engine takes slightly more time to realize a wrong turn due to a system call.

Diff

We now consider the *diff* utility. *Diff* is more challenging than the *uServer* for our dynamic analysis, as its behavior depends more heavily on input. Moreover, *diff* generates very long constraint sets, placing a heavy burden on the constraint solver. For these reasons, dynamic analysis attains a coverage of only 20% of branches, during 1 hour of symbolic execution. In total, there are 8840 branches in the program. *dynamic* identifies 440 of them as being symbolic, *static* 4292, and *dynamic+static* 3432.

Instrumentation overheads. We run *diff* on two sample text files. Figure 5.6 shows the CPU time of the four configurations of our system, normalized against the non-instrumented execution. Consistent with our prior results, *dynamic* and *dynamic+static* perform best with an overhead of approximately 35%.

Version		# of symbolic branch locations logged / corresponding # of executions		# of symbolic branch locations NOT logged / corresponding # of executions	
		LC	HC	LC	HC
Exp. 1	dynamic	40 / 722	43 / 725	8 / 173	5 / 170
	dynamic+static	40 / 722	43 / 725	8 / 173	5 / 170
	static	49 / 871		0	
	all branches	49 / 465		0	
Exp. 4	dynamic	43 / 245017	65031 (61)	21 / 107650	7 /1950
	dynamic+static	64 / 88739	64612 (67)	3 / 7397	1 / 1021
	static	50 / 39581		0	
	all branches	50 / 37346		0	

Table 5.8: Number of symbolic branch locations and symbolic branch executions logged and not logged for two input scenarios with the *uServer* when not logging system call results.

Version	**Exp. 1**	**Exp. 2**
dynamic	∞	∞
dynamic + static	1s	12s
static	1s	12s
all branches	1s	12s

Table 5.6: Time in seconds needed to reproduce two input scenarios to *diff*. The infinity symbol means that the experiment did not terminate in one hour.

Version		**symbolic branch locations logged / corresponding executions**	**symbolic branch locations NOT logged / corresponding executions**
Exp. 1	dynamic	3 / 2125686	32 / 2369765
	dyn.+static	13 / 904	0
	static	13 / 904	0
	all branches	13 / 904	0
Exp. 2	dynamic	3 / 2478280	24 / 2102506
	dyn.+static	21 / 54623	0
	static	21 / 54623	0
	all branches	21 / 54623	0

Table 5.7: Number of symbolic branch locations and symbolic branch executions logged and not logged for two input scenarios to *diff*.

Reproducing bugs. We replay two executions of *diff* comparing relatively small but different text files. Tables 5.6 and 5.7 list the results of these experiments. Because the coverage obtained during dynamic analysis is relatively low, the *dynamic* configuration is unable to finish the experiments in 1 hour. The few tens of unlogged symbolic branch locations quickly create a very large number of paths to explore, making it impossible for this approach to finish within the allotted time. In contrast, the three other configurations, and in particular *dynamic+static*, do not suffer from any unlogged symbolic branches and therefore replay quickly.

Collectively, these results again demonstrate that *dynamic+static* strikes the best balance between instrumentation overhead and bug reproduction time.

5.6 Discussion and Future Work

Branch coverage. Our current results suggest that using a branch trace, even partial, is effective at reducing the amount of searching needed to reproduce a specific buggy execution path. To maintain low instrumentation overhead, it is necessary to obtain sufficient coverage with the initial symbolic execution phase. This problem has received attention in the literature ([10, 26]), and significant progress has been made in recent years. In chapter 4 we describe our own efforts to improve the coverage of upgraded code. While it is not always possible to achieve 100% coverage, this is a typical goal when testing an application *prior to shipping*. Therefore, the testing effort can be leveraged to identify the symbolic branches at the same time. Moreover, manual test cases can be used in conjunction with symbolic execution to boost code coverage, and many applications already have test suites covering most of their codes.

Constraint solving. Symbolic execution is limited by the ability to solve the resulting constraints. In particular, certain types of programs generate constraints that current state of the art solvers cannot solve. Constraint solving is an active research topic and our approach should directly benefit from any advances in this field.

Non-determinism. Two approaches exist for dealing with non-determinism, resulting, for instance, from system calls or random number generators. One can either log the outcome of the non-deterministic event or one can treat it as (symbolic) input during replay. In this chapter, we strike a middle ground between these approaches, logging the outcome of non-deterministic events that are likely to cause a great deal of search during replay. The results in Section 5.5 validate this approach, but a more comprehensive treatment of non-deterministic events could be explored.

Multithreading. We can extend our system to support multi-threaded applications by modifying it in two ways. First, the branch trace needs to be split into multiple traces, one per thread. Second, the ordering of thread execution needs to be recorded as well. Implementing the first modification is trivial, and is unlikely to impose any significant additional overhead. The second modification is more difficult. Others [1, 58] have experimented with ideas for recreating a suitable thread scheduling to find race conditions. Our approach of logging a partial trace of branches is complementary to those efforts and could considerably speed up the replay of multi-threaded programs with races.

Long-running applications. The storage overhead and replay time of long-running applications can be problematic. Consider, for example, the case of a Web server running for weeks before crashing. Our current approach reduces storage overhead as much as possible, but with these applications this overhead may still be high. Furthermore, replaying such a long trace may be infeasible, as it will be longer than the original run. Pushing the concept of a partial branch trace further, our approach could be extended to simplify this problem by implementing support for checkpointing. An instrumented application could periodically take a checkpoint of its state and discard the current branch log. Logging branches would then continue from the checkpoint only. The checkpoint would include enough information on the data structures of the program (but not its content). With this information, a symbolic execution engine can treat their content as symbolic, and replay the branch log starting from there. We leave the implementation and the associated research questions of the checkpointing mechanism for future work.

Concolic vs. symbolic. The particular form of symbolic execution we use in this chapter is called concolic execution [51]. The main difference from pure symbolic execution (as in [10], for instance) is that the engine repeatedly executes the program from beginning to end with concrete inputs, instead of exploring multiple paths in parallel. This implementation difference has no fundamental impact on our system, as in both cases the engine can select the paths in the same order. On one hand, the fact that the application is rerun from the beginning for every path imposes some additional overhead. On the other hand, because concrete inputs are always used, it makes the work of the solver easier. In many instances, branch conditions are already satisfied by the random input chosen. To the best of our knowledge, no comparative studies of the impact of the different implementations have been published yet.

5.7 Conclusion

In this chapter, we consider the problem of instrumenting programs to reproduce bugs effectively, while keeping user data private. In particular, we focus on the tradeoff between the instrumentation overhead experienced by the user and the time it takes the developer to reproduce a bug in the program.

We explore this tradeoff by studying approaches for selecting a partial set of branches to log during the program's execution in

the field. Specifically, we propose to use static analysis (dataflow and points-to analysis) and/or dynamic analysis (time-constrained symbolic execution) to find the branches that depend on input. Our instrumentation methods log only those branches to limit the instrumentation overhead. When a user encounters a bug, the developer uses the partial branch log to drive a symbolic execution engine in efficiently reproducing the bug.

Our results show that the instrumentation method that combines static and dynamic analyses strikes the best compromise between instrumentation overhead and bug reproduction time. For the programs we consider, this combined method reduces the instrumentation overhead by up to 92% (compared to using static analysis only), while limiting the bug reproduction time (compared to using dynamic analysis only).

We conclude that our characterization of this tradeoff and our combined instrumentation method represent important steps in improving bug reporting and optimizing symbolic execution for bug reproduction.

Chapter 6

Related Work

6.1 Characterizing Upgrades

As far as we know, only two other works characterized upgrades. Beattie *et al.* [5] tried to determine when it is best to apply security patches. They built mathematical models of the factors affecting when to patch, and collected empirical data to validate the model. Relative to Mirage, the paper focused solely on security patches and did not consider the benefits of user-machine testing or staged deployment.

The study by Gkantsidis *et al.* [23] focused on the Windows environment and on the networking aspects of deploying upgrades. Most importantly, the study did not consider several important issues, including upgrade installation and testing, upgrade problems, and problem reporting. Our survey addressed all of these issues.

6.2 Upgrade Deployment

Deployment Strategy

As far as we know, no previous work has considered staged upgrade deployment. In fact, the two previous works on large-scale upgrade deployment focused solely on (1) deploying upgrades as quickly as possible to all users while reducing the load on the vendor's site [23]; and (2) creating a deployment infrastructure that can be tailored by explicit contracts between vendors and users [52]. In contrast to (1), Mirage recognizes the tradeoff between the speed of deployment and the likelihood of testing problems at the user machines. In contrast to (2), Mirage does not consider contracts, since they are rare in practice, especially in the open-source community.

Commercial upgrade (patch) management systems and tools, e.g., [32, 47], deploy upgrades within enterprises. They typically concentrate on discovering machines on the enterprise's network, assessing which upgrades are needed, and applying the patches while minimizing the number of reboots. Some providers, such as [47], manually collect upgrades from vendors and test them locally before sending them to their customer enterprises. Relative to Mirage, these upgrade-management systems do not help individual users and have no automatic support for testing upgrades after they are applied to the machines in the enterprise. Nevertheless, the providers that manually test upgrades do produce a primitive form of staging for enterprises. However, they are unlike representatives (or beta testers), since they have no relationship to the vendor. In fact, their existence does not change the fact that vendors deploy their upgrades to the entire world as a single huge cluster.

Clustering Machines

The Pastiche [17] peer-to-peer backup system builds a structured overlay by replacing the proximity metric with a measure of similarity between sets of neighbors' files. Although Pastiche does not explicitly cluster machines, one could envisage Pastiche being used to perform distributed clustering based on the list of environmental resources at each machine.

Others have considered surviving Internet catastrophes by backing up data on dissimilar hosts. This problem roughly corresponds to the problem of recovering from a globally-deployed problematic upgrade. In Phoenix [31], Junqueira *et al.* observe that backups should be done on hosts that do not share the same potential vulnerabilities. Interestingly, the clustering of dissimilar hosts in Phoenix has the opposite goal to our clusters, which cluster similar hosts. Furthermore, Mirage clusters machines based on different characteristics than Phoenix.

Integrating Deployment, Testing, and Reporting

At a high-level, the most similar work to Mirage is a position paper by Cook and Orso [14], which also proposes to integrate upgrade deployment, user-machine testing (via white-box approaches), and problem reporting back to developers. However, they did not consider staged deployment or any type of clustering. Furthermore, they did not attempt to characterize upgrades or evaluate their proposed system.

6.3 Testing

Static Analysis and Model Checking

Static analysis [4, 6] can automatically check a number of useful properties in the code, but it can miss many important problems. Moreover, it may generate many unnecessary warnings.

Model checking of implementations [38, 37, 57] can be used to find bugs in a systematic fashion. However, symbolic execution overcomes the need for creating special testing harnesses that are needed for model checking.

Testing and Dynamic Analysis

Testing plays an important role in ensuring the overall quality of the software, because it aims to detect errors in program logic, check boundary values, and provide high code coverage. However, it is very hard to achieve good coverage in practice, as it requires careful choice of inputs. Oasis uses concolic execution to generate input, and implements a new heuristic to optimize the coverage of new or affected code when regression testing.

Dynamic analysis tools (such as Valgrind [43]) can provide valuable help to detect bugs during testing. Unfortunately, they detect bugs only on the paths on which they are executed. PathExpander [34] improves path coverage of such tools by selectively executing non-taken paths in a sandbox.

In contrast, Oasis uses dynamic analysis to detect bugs, and concolic execution to improve coverage of the test cases.

Symbolic and Concolic Execution

There has been a large body of work on using symbolic and concolic execution for automated test generation [7, 8, 11, 10, 16, 15, 19, 25, 27, 26, 35, 49, 51, 9].

Most of the work in this area tries to address the two main problems of symbolic execution: dealing with the environment [11, 10] and path explosion [7, 11, 10, 19, 25, 35]. Most recently, KLEE [10] uses environment modeling, and aggressive constraint caching and search heuristics to deal with these two problems, respectively. As a result, KLEE has been successful in identifying bugs in heavily debugged code. Similar to previous works, Oasis provides models for system calls. To address the problem of path explosion, Oasis relies on search heuristics, and proposes interactive symbolic execution as a way to advance the state of the art in this area.

Other symbolic and concolic execution engines [10, 9] implement path exploration heuristics that try to drive the program into parts of the code that have not been explored yet. They typically use a statically computed control flow graph and use it to try to follow branches that are more likely to drive the application to yet unexplored parts of the code. Oasis builds on those ideas and extends them by analyzing the source code to pinpoint the new code in upgrades, and tracks the effect of this new code on the rest of the code at runtime.

Existing approaches to concolic execution, e.g., DART [27], can use random inputs to drive code execution along any given path. However, random inputs have low probability of passing through a series of input validation checks that typically exist in software. In contrast, Oasis uses any valid inputs to pass through these checks and quickly start exploring deep code paths.

Symbolic Java PathFinder [49] uses system-level concrete execution to reach a unit of code, and then uses unit-level symbolic execution. In contrast with this work, Oasis does not require manual validation of potentially erroneous inputs (that are the result of isolated unit testing).

Finally, relative to the existing body of work on symbolic and concolic execution, Oasis leverages the differences in the code to more quickly explore paths leading to the new or changed code. Doing so reduces the time needed for regression testing, for example in the case of software upgrades.

6.4 Bug Reporting

At one end of the spectrum between instrumentation overhead and bug reproduction effort are record-replay systems that try to capture the interactions between the program and its environment. Different systems capture interactions at different levels. Most systems capture them at the system call or library level. For example, ReVirt [18] logs interactions at the virtual machine level. All record-replay systems suffer from the overhead of logging the interactions. To reduce this overhead, R2 [29] asks the developer to manually specify at what interfaces to capture the program's interactions with its environment.

At the other end of the spectrum are conventional bug reporting systems, which provide a coredump, but no indication of how the program arrived at the buggy state. Obviously, there is no recording overhead, but it takes considerable manual search to find out how

the problem came about.

ESD (Execution Synthesis Debugger [58]) is an attempt to automate some of that search. It tries to do so without recording any information about the program execution. Instead, it uses the stack trace at the time of the program crash, and symbolically executes the program to find the path to the bug location. Although it uses static analysis and other optimizations to reduce the number of paths it needs to explore, it remains fundamentally limited by the exponential path explosion of symbolic execution. Our approach instead performs logging of a set of judiciously chosen branches. This allows us to speed up the automated search with limited runtime overhead. As our results with the coreutils show, our methods reproduce bugs faster, albeit at some modest cost in runtime.

BBR (Better Bug Reporting, [12]) investigates the same tradeoff, although more from the perspective of maintaining privacy when a bug is reported to the developer. During execution it logs the program's inputs. After a crash, it replays the entire execution on the user machine based on those inputs. Replay uses an instrumented version of the program that collects the constraints implied by the direction of the branches taken in the program. An input set that satisfies these constraints is then returned to the developer. Unlike BBR, we do not log the users' inputs, or require whole-program replay and the execution of a constraint solver on the user machine. Instead, we incur limited logging overhead at the user site and some exploration of alternative paths at the developer site.

Another approach for maintaining privacy is explored by the Panalyst system [56]. After a runtime error, the user initially reports only public information. The developer tries to exploit this information using symbolic execution, but can query the user for additional information. The user can choose whether or not to respond to the developer queries. In the limit, Panalyst's effectiveness is constrained by the exponential cost of symbolic execution.

Triage [54] explores yet another way of debugging errors at user sites. It periodically checkpoints programs, and after a failure, it restarts the program from a checkpoint. Heavyweight instrumentation, exploration of alternatives (delta-debugging), and speculative execution may be used during replay. Some applications were successfully debugged using this approach, but the checkpoint may have to be far back in time to allow meaningful exploration.

The same tradeoff between instrumentation overhead and bug reproduction time has also been explored for debugging multithreaded programs. To faithfully replay the execution of a thread, the shared

memory interactions with other threads need to be logged. The cost of doing so is very high, and therefore the PRES [46] debugger selectively omits logging certain interactions, but requires multiple replay runs before it can recreate a path to a bug. Similarly, in order to avoid logging all shared memory accesses, ODR [1] allows some degree of inconsistency between the actual execution and the replay, provided that the inconsistencies do not affect the output of the program. The techniques used by PRES and ODR could be combined with partial logging of branches as presented in this paper.

Logging of branches has been used to report a program's behavior before a bug in Traceback [3]. The system uses this behavior to reconstruct the control flow leading to the problem. To reduce the instrumentation overhead, Traceback uses static analysis to minimize the number of instructions required to instrument branches, and only logs the most recent branches. Our system goes further by combining dynamic and static analyses to reduce the number of instrumented branches, and reproducing the entire path to the bug.

Chapter 7

Conclusion

In this dissertation, we study the problem of improving the quality and reliability of software upgrades, and reduce the impact of problems.

We motivate our work using a survey of system administrators. The results confirm that upgrades are done frequently, that problems are quite common, that these problems can cause severe disruption and that therefore upgrades are often delayed, and that users seldom fully report the problems to the vendor.

To improve on the current situation, we argue the necessity to revise the entire software upgrade process. To this end, we present Mirage, a framework that integrates testing, deployment and bug reporting in an integrated system.

Mirage's deployment subsystem allows the vendor to deploy its upgrades in stages over clusters of users sharing similar environments. Our evaluation demonstrates that our clustering algorithm achieves its goal of separating machines with dissimilar environments in different clusters, while not creating an impractically large number of clusters. A simulation study of two staged deployment protocols demonstrates that staging significantly reduces upgrade overhead, while still achieving low deployment latency. Furthermore, a suitable choice of protocol allows a vendor to achieve different objectives.

Oasis, the testing subsystem of Mirage, improves on current state-of-the-art concolic and symbolic engines by implementing an new heuristic to prioritize the exploration of new or affected code in the upgrade. Furthermore, we propose interactive symbolic execution, a new approach exposing the problem of path exploration to the tester using a graphical user interface. The tester can use interactive symbolic execution as a learning tool to develop new heuristics or as a complement to existing heuristics to manually

influence the exploration and steer it towards interesting areas of the code. Our preliminary results indicate that our heuristic successfully explores the new or affected code more intensively than existing techniques and shows that interactive symbolic execution is a promising technique to advance the state of the art in search heuristics.

In spite of all of these efforts, some bugs are bound to remain in the software when it is deployed, and will be discovered and reported only later by the users. With the last component of Mirage, we consider the problem of instrumenting programs to reproduce bugs effectively, while keeping user data private. In particular, we focus on the tradeoff between the instrumentation overhead experienced by the user and the time it takes the developer to reproduce a bug in the program.

Specifically, we propose to use static analysis (dataflow and points-to analysis) and/or dynamic analysis (time-constrained symbolic execution) to find the branches that depend on input. Our instrumentation methods log only those branches to limit the instrumentation overhead. When a user encounters a bug, the developer uses the partial branch log to drive a symbolic execution engine to efficiently reproduce the bug.

Our results show that the instrumentation method that combines static and dynamic analyses strikes the best compromise between instrumentation overhead and bug reproduction time. For the programs we consider, this combined method reduces the instrumentation overhead by up to 92% (compared to using static analysis only), while limiting the bug reproduction time (compared to using dynamic analysis only).

We conclude that our characterization of this tradeoff and our combined instrumentation method represent important steps in improving bug reporting and optimizing symbolic execution for bug reproduction.

By combining up-front testing, staged deployment, testing on user machines, and efficient reporting, Mirage successfully reduces the number of problems, minimizes the number of users affected, and shortens the time needed to fix remaining problems.

Survey about software upgrades and associated problems

In the scope of our research project, we have designed this survey about software upgrades. It is aimed at understanding the software upgrading process as well as collecting testimonies about most common issues.

Please note that our research project is a joint effort from two different laboratories at EPFL (http://labos.epfl.ch and http://nsl.epfl.ch in Switzerland). The project is not affiliated with or sponsored by any commercial organization. The results of our survey will be used to evaluate a new infrastructure aimed at improving software deployment and testing. We will share the survey results with the practitioner and research communities through scientific papers.

In order to recognize your effort in providing the testimony, we will hold a lottery to select four winners who will each receive a $50 (50 american dollars) amazon.com gift certificate.

It should not take you more than 10 to 15 minutes to complete the survey. We thank you very much for your participation !

If you have problems with this form, please send us an email at olivier.crameri@epfl.ch

[more information...]

* mandatory field

* **What is your experience in performing system administration ?**

- ○ 0-1 yr
- ○ 2-5 yr
- ○ 5-10 yr
- ○ > 10 yr

* **What is your level of responsibility as a system administrator ?**

- ○ I run the system alone
- ○ I run the system and a few people help me
- ○ I along with few others are the main people in-charge
- ○ I assist other senior administrators, and manage in their absence
- ○ Other, please explain

* **How many machines are you responsible for ?**

- ○ less than 10
- ○ 10 to 20
- ○ 20 to 50
- ○ more than 50

* **What best describes the types of machines you are administering ?**

- ○ Only desktop systems
- ○ Desktops and servers
- ○ Only servers
- ○ Other, please explain:

* **Which operating systems are you administering ?**

- ☐ Windows on desktops (XP, 2000,...)
- ☐ Windows server
- ☐ Linux
- ☐ FreeBSD
- ☐ Mac OS X
- ☐ HP-UX
- ☐ AIX
- ☐ Solaris
- ☐ IBM i5/OS (OS/400)
- ☐ Other, please specify:

The rest of the survey is about software upgrades. By software upgrade, we mean acquiring a new version of an application, an improved software module or component, or simply a "patch" to fix a bug, and integrating it into the system.

* **How frequently do you install a software upgrade ?**

- More than once a week
- Once a week
- Once every couple of weeks
- Once a month
- Once per quarter
- Once per semester
- Once a year
- Not even once a year

* **Do you use any software (e.g. package management system or OS-built in upgrade system) to install your upgrades ?**

- No
- Yes, please specify which one:

* **Please rank what are the most important reasons to apply an upgrade** insert a value from 1 (most important) to 5 (least important) for each item

- Bug fix
- Security fix
- New feature
- User requet
- Other(s) — Please specify and rank

* **Do you have a strategy to test upgrades before applying them ?**

- No
- Yes, please explain:

* **In general, did you encounter problems with software upgrades ?**

- No, in general it works out of the box
- No, thanks to the strategy I use to test software upgrades
- Yes, but I usually manage to fix them quickly
- Yes, and sometimes it is catastrophic
- If none of the answer above apply to you, please explain:

* **In your opinion, what is the percentage of upgrades that cause problems ?** (Please enter a number between 1 and 100. For instance 1% => 1 upgrade problem for every 100 problem-free upgrades).

* **Do you ever refrain from installing an upgrade because you expect problems ?**

- No
- Yes. In this case, please explain when/how you decide that it is safe to upgrade:

The rest of this survey assumes that you've had problems with software upgrade in the past. If it is not the case, you can jump to the end of the survey. If, however, you've had problems, we would highly appreciate if you could provide us with as many details as possible by answering the following questions.

We would like to draw your attention to the fact that this is the most important part of the study for us. Describing accurately examples of upgrade problems is what will allow us to reproduce and understand them in order to make progress with our prototype

Do you have one or more examples of software upgrade(s) that caused problem(s) ?
We're interested here in consequences of applying the upgrade itself, i.e. problems that happen AFTER the upgrade has been successfully installed. We're not interested here in problems happening during the installation (such as package conflicts, dependency hell and so on and so forth...) Please describe as many examples as you can think of in the following four boxes.

Please describe the consequence of the problem(s)

Please tell us with which software(s) you had the problem(s). Please include version number(s)

And on what OS(s)/distribution(s) and kernel(s) were they ?

What solution(s) did you find to fix the problems ?

When you do encounter an upgrade problem, how often do you report it back to the distributor/vendor ?

- **Every time**
- **Most of the time**
- **Rarely**
- **Never**

When you do report back a problem to the distributor/vendor, what type of information do you usually send ?

If one produced software that automatically tested upgrades on their local machines with good accuracy (i.e. good upgrades pass the automatic testing whereas bad upgrades fail it), would you install and use it ?

- **Yes**
- **No, why ?**

Please rank what you think are the most common causes for problems in software upgrades:
insert a value from 1 (most important) to 5 (least important)

The upgrade is simply buggy		
Important feature/behavior was removed/altered		
Legacy issue		*(such as deprecated api, misuse of api,...)*
Broken dependency		*(i.e. installing the upgrade cause problem with a dependent software)*
Packaging issue		*(i.e. the upgrade itself is "bugfree" except for the packaging)*
Other(s)	**Please explain and rank**	

Out of 100 problematic upgrades, could you please estimate how many of them are causing problems during installation ?
(this is by opposition to the number of upgrades that cause problem AFTER installation, i.e. the ones that are difficult to install because of conflicts, missing dependency, version mismatch,...)

*	**Please insert your email address** We will use the email address to contact you if we need additional information and if you win the lottery. We won't share your email address with any third party.	

Send

CURRICULUM VITAE

Olivier Crameri
Swiss and Belgian citizen
Date of birth: 28th march 1981
Email: olivier.crameri@a3.epfl.ch

ch. du Cheminet 16
1162 Saint-Prex
Switzerland
Tel: +41 76 394 28 90

PhD. in Computer Science, EPFL

Education:

August 2005 – June 2011:	EPFL, Lausanne PhD. in Computer Science. Developed and implemented novel techniques for improving the quality and reliability of software.
June 2009, June 2010:	Rutgers University, Piscataway, New Jersey: Visiting student, supervised by Professor Ricardo Bianchini. Work on testing techniques for software upgrades.
April 2005 – July 2005:	Rice University, Houston, Texas: Visiting student, supervised by Professor Alan L. Cox. Work on improving the performance of virtual memory in the FreeBSD operating system.
October 1999 – March 2005:	EPFL, Lausanne Master of Science in Computer Science. Diploma project on asynchronous I/O in Linux.

Professional experience:

July 2000 – today:	Net Oxygen Sàrl (www.netoxygen.ch). Co-founder, project manager and developer. Lead architect and developer of Net Oxygen's web hosting platform and billing system. Currently on the board of directors.
April 2006 – September 2006:	Hewlett-Packard Laboratories, Palo Alto, California: Research intern, working on I/O virtualization.

Selected publications:

- Striking a New Balance Between Program Instrumentation and Debugging Time.
  EuroSys '11: Proceedings of the 6th European Conference on Computer Systems, 2011, with R. Bianchini and W. Zwaenepoel.
- Toward Online Testing of Federated and Heterogeneous Distributed Systems.
  Proceedings of The 2011 USENIX Annual Technical Conference, 2011 (to appear), with M. Canini, V. Jovanovic, D. Venzano, B. Spasojevic and D. Kostic.
- Oasis: Concolic Execution Driven by Test Suites and Code Modifications.
  Technical report, EPFL, 2009, with R. Bachwani, T. Brecht, R. Bianchini and W. Zwaenepoel.
- Mirage, an Integrated Software Upgrade Testing and Distribution System.
  Proceedings of The 21st ACM Symposium on Operating Systems Principles (SOSP), 2007, with N. Knezevic, D. Kostic, R. Bianchini and W. Zwaenepoel.

Bibliography

[1] Altekar, G., and Stoica, I. ODR: Output-deterministic Replay for Multicore Debugging. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), ACM, pp. 193–206.

[2] ASF Bugzilla Bug 10073 upgrade from 1.3.24 to 1.3.26 breaks include directive. http://issues.apache.org/bugzilla/show_bug.cgi?id=10073.

[3] Ayers, A., et al. TraceBack: First Fault Diagnosis by Reconstruction of Distributed Control Flow. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM, pp. 201–212.

[4] Ball, T., and Rajamani, S. K. The slam project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2002), ACM, pp. 1–3.

[5] Beattie, S., Arnold, S., Cowan, C., Wagle, P., Wright, C., and Shostack, A. Timing the Application of Security Patches for Optimal Uptime. In *Proceedings of the 16th Systems Administration Conference* (2002).

[6] Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R. The software model checker blast. *STTT 9*, 5-6 (2007), 505–525.

[7] Boonstoppel, P., Cadar, C., and Engler, D. R. Rwset: Attacking path explosion in constraint-based test generation. In *TACAS* (2008), pp. 351–366.

[8] Brumley, D., Newsome, J., Song, D., Wang, H., and Jha, S. Towards automatic generation of vulnerability-based

signatures. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 2–16.

[9] Burnim, J., and Sen, K. Heuristics for scalable dynamic test generation. Tech. Rep. UCB/EECS-2008-123, EECS Department, University of California, Berkeley, Sep 2008.

[10] Cadar, C., Dunbar, D., and Engler, D. R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating systems design and implementation* (2008), pp. 209–224.

[11] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R. Exe: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security* (New York, NY, USA, 2006), ACM, pp. 322–335.

[12] Castro, M., Costa, M., and Martin, J.-P. Better Bug Reporting with Better Privacy. In *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ACM, pp. 319–328.

[13] Chaum, D. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM 4*, 2 (February 1981).

[14] Cook, J., and Orso, A. MonDe: Safe Updating through Monitored Deployment of New Component Versions. In *Proceedings of the 6th Workshop on Program Analysis for Software Tools and Engineering (PASTE)* (September 2005).

[15] Costa, M., Castro, M., Zhou, L., Zhang, L., and Peinado, M. Bouncer: Securing Software by Blocking Bad Input. In *SOSP* (2007).

[16] Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., and Barham, P. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th SOSP* (October 2005).

[17] Cox, L. P., Murray, C. D., and Noble, B. D. Pastiche: Making Backup Cheap and Easy. In *Proceedings of the 5th*

Symposium on Operating Systems Design and Implementation (December 2002).

[18] Dunlap, G. W., et al. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 211–224.

[19] Emmi, M., Majumdar, R., and Sen, K. Dynamic test input generation for database applications. In *ISSTA* (2007), pp. 151–162.

[20] Fixing A Troubled Firefox 2.0 Upgrade. http://softwaregadgets.gridspace.net/2006/10/30/fixing-a-troubled-firefox-20-upgrade/.

[21] Firefox crashes after 1.5.0.9 update. http://www.ubuntuforums.org/showthread.php?t=331274.

[22] Ganesh, V., and Dill, D. L. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)* (Berlin, Germany, July 2007), Springer-Verlag.

[23] Gkantsidis, C., Karagiannis, T., Rodriguez, P., and Vojnovic, M. Planet Scale Software Updates. In *Proceedings of SIGCOMM* (September 2006).

[24] Glerum, K., et al. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), ACM, pp. 103–116.

[25] Godefroid, P. Compositional dynamic test generation. In *POPL* (2007), pp. 47–54.

[26] Godefroid, P., Kiezun, A., and Levin, M. Y. Grammar-based whitebox fuzzing. In *PLDI* (2008), pp. 206–215.

[27] Godefroid, P., Klarlund, N., and Sen, K. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM, pp. 213–223.

[28] Godefroid, P., Levin, M. Y., and Molnar, D. Automated whitebox fuzz testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)* (2008).

[29] Guo, Z., et al. R2: an Application-level Kernel for Record and Replay. In *OSDI'08: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), USENIX Association, pp. 193–208.

[30] Heyer, L. J., Kruglyak, S., and Yooseph, S. Exploring Expression Data: Identification and Analysis of Coexpressed Genes. In *Genome Research* (1999), pp. 1106–1115.

[31] Junqueira, F., Bhagwan, R., Hevia, A., Marzullo, K., and Voelker, G. M. Surviving Internet Catastrophes. In *Proceedings of the USENIX 2005 Annual Technical Conference* (April 2005).

[32] Kaseya Patch Management. http://www.kaseya.com/products/patch-management.php.

[33] Kiezun, A., Ganesh, V., Guo, P. J., Hooimeijer, P., and Ernst, M. D. Hampi: a solver for string constraints. In *Proceedings of the eighteenth international symposium on Software testing and analysis* (New York, NY, USA, 2009), ISSTA '09, ACM, pp. 105–116.

[34] Lu, S., Zhou, P., Liu, W., Zhou, Y., and Torrellas, J. Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 38–52.

[35] Majumdar, R., and Sen, K. Hybrid concolic testing. In *ICSE* (2007), pp. 416–426.

[36] Mosberger, D., and Jin, T. httperf: A Tool for Measuring Web Server Performance. In *The First Workshop on Internet Server Performance* (Madison, WI, June 1998), pp. 59—67.

[37] Musuvathi, M., and Engler, D. R. Model Checking Large Network Protocol Implementations. In *NSDI* (2004).

[38] Musuvathi, M., Park, D. Y. W., Chou, A., Engler, D. R., and Dill, D. L. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 75–88.

[39] Muthitacharoen, A., Chen, B., and Mazieres, D. A Low-bandwidth Network File System. In *Proceedings of the 18th SOSP* (December 2001).

[40] Myers, G. J. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[41] Report of PHP problem after MySQL upgrade. http://www.linuxquestions.org/questions/showthread.php?t=425535.

[42] Necula, G. C., et al. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of Conference on Compilier Construction* (2002).

[43] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42*, 6 (2007), 89–100.

[44] Openssh, a free version of the ssh connectivity tools. http://www.openssh.com/.

[45] Pariag, D., et al. Comparing the Performance of Web Server Architectures. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), ACM, pp. 231–243.

[46] Park, S., et al. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 177–192.

[47] PatchLink. http://www.patchlink.com/.

[48] PHP5 Migration guide. http://ch2.php.net/manual/en/migration5.incompatible.php.

[49] Păsăreanu, C. S., Mehlitz, P. C., Bushnell, D. H., Gundy-Burlet, K., Lowry, M., Person, S., and Pape, M. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *ISSTA* (2008).

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

[51] Sen, K., Marinov, D., and Agha, G. CUTE: a Concolic Unit Testing Engine for C. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference* (New York, NY, USA, 2005), ACM, pp. 263–272.

[52] Sobr, L., and Tuma, P. SOFAnet: Middleware for Software Distribution over Internet. In *Proceedings of the IEEE Symposium on Applications and the Internet* (January 2005).

[53] Srivastava, A., and Thiagarajan, J. Effectively prioritizing tests in development environment. *SIGSOFT Softw. Eng. Notes 27*, 4 (2002), 97–106.

[54] Tucek, J., et al. Automatic On-line Failure Diagnosis at the End-user Site. In *HOTDEP'06: Proceedings of the 2nd Conference on Hot Topics in System Dependability* (Berkeley, CA, USA, 2006), USENIX Association, pp. 4–4.

[55] The uClibc Library, a C Library for Linux. http://www.uclibc.org/.

[56] Wang, R., Wang, X., and Li, Z. Panalyst: Privacy-aware Remote Error Analysis on Commodity software. In *SS'08: Proceedings of the 17th Conference on Security Symposium* (Berkeley, CA, USA, 2008), USENIX Association, pp. 291–306.

[57] Yang, J., Twohey, P., Engler, D., and Musuvathi, M. Using Model Checking to Find Serious File System Errors. *ACM Trans. Comput. Syst. 24*, 4 (2006), 393–423.

[58] Zamfir, C., and Candea, G. Execution Synthesis: a Technique for Automated Software Debugging. In *EuroSys '10: Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), ACM, pp. 321–334.