If It Ain't Broke, Don't Fix It: Challenges and New Directions for Inferring the Impact of Software Patches

Jon Oberheide, Evan Cooke, Farnam Jahanian
Electrical Engineering and Computer Science Department
University of Michigan, Ann Arbor, MI 48109
{jonojono, emcooke, farnam}@umich.edu

Abstract

Software patches are designed to have a positive effect on the operation of software systems. However, these patches may cause incompatibilities, regressions, and other unintended negative impact on the reliability, performance, and security of software. In this paper, we propose PatchAdvisor, a technique to improve the manageability of the patching process for administrators by automatically inferring the impact of a patch or upgrade. PatchAdvisor inspects a software system and its patch using a combination of static control flow analysis, dynamic execution traces, and ranking heuristics to automatically infer the potential impact of the patch. To evaluate the feasibility of our approach, we implement an initial prototype of PatchAdvisor using the IDA and PaiMei frameworks and demonstrate its effectiveness on a real-world web application stack. Finally, we discuss the challenges and future research directions in this problem domain.

1 Introduction

Modern software systems are complex. These systems consist of many components and layers that interact with and depend on each other. Rich web application stacks, custom in-house enterprise applications, and commercial off-the-shelf software are commonplace, yet are often of considerable complexity. Due to their complexity, these software systems frequently require patching and updating to fix bugs, patch security vulnerabilities, add functionality, or increase performance.

For example, we surveyed several Gentoo Linux machines used for various purposes in our research group: a workstation, a server used for data processing and research experiments, and a server used to host simple web content and revision control. Each host had 923, 655, and 192 software packages installed respectively for a total of 1453 unique packages. Of those 1453 packages, there were 2402 upstream updates to those packages during the 2008 calendar year.

While patches are intended to have a positive impact on the operation of a software component, such upgrades are not without risk, and updates can have unwanted consequences. The number of potential interactions between the various components, which are often developed and updated independent of each other, of a software system of any considerable scale is enormous. Misplaced assumptions between the operation of these components may lead to bugs, incompatibilities, regressions, and other unintended negative effects on the reliability, performance, and security of the software system.

As downtime and failures can have significant operational and financial impact in modern systems, these patches must be applied with caution and prudence. Traditionally, the adage often repeated in the domain of software reliability is: "If it ain't broke, don't fix it". That is, upgrades and patches are often unnecessary and unwise if the software appears to be functioning correctly. However, blindly ignoring important patches may be an unwise policy as well. In the real world, it is often the task of a system's administrator to manage and the gauge the risk involved in upgrading a particular software component. While existing approaches, such as test suites, may assist an administrator, we believe that significant advancements are possible that improve the manageability of the patching process, which thereby can positively influence factors such as reliability, security, and survivability [3].

We propose an approach, called PatchAdvisor, whose goal is to automatically infer the impact that a patch will have on a software system. This information can be of great value to administrators in charge of software upgrades, allowing them to make more informed decisions during the patch process. Our approach is based on the simple observation that patching an area of the software's code that is frequently used during execution is likely to have a greater impact, whether positive or negative, on

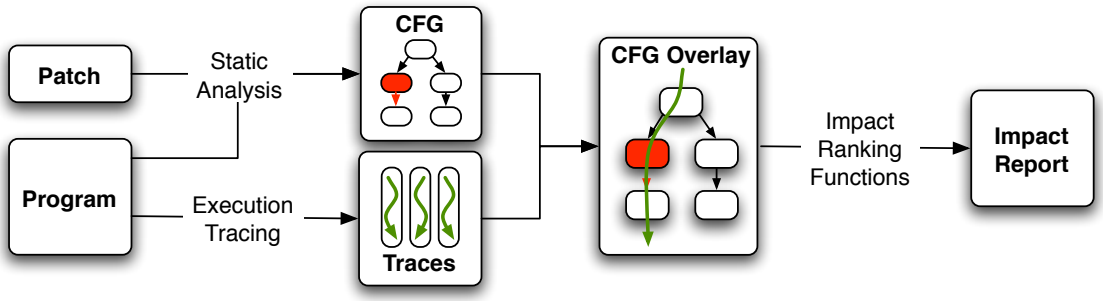

Figure 1: An overview of the PatchAdvisor architecture.

the operation of the software. PatchAdvisor combines static analysis of the candidate software and associated patch, dynamic execution tracing of the software in its normal operating environment, and heuristic functions to automatically infer the potential impact of the patch.

We present the PatchAdvisor approach, describe our prototype implementation built using the IDA [9] and PaiMei [13] frameworks, and perform an evaluation using a real-world web application. Beyond our initial investigation, we believe there is much progress to be made in this research area and discuss several of the challenging problems worthy of further investigation.

2 Existing Approaches

Maximizing the positive benefits of an upgrade or patch while minimizing the risk of adverse affects is a difficult problem. However, there are existing approaches that may assist in lessening the difficulty for an administrator. One popular approach currently employed by administrators is the use of test suites. For example, before deploying an upgraded version of a software component, an administrator can run test cases against the new version to ensure that the test case output matches the expected results. While the use of test suites is a valuable tool in the administrator's toolbox when dealing with the testing and deployment of patched or upgraded software, maintaining comprehensive test suites can be expensive and the resulting code coverage of test cases may be inadequate [1]. We feel that PatchAdvisor is complimentary to existing software test suites and can provide valuable insight in cases where testing coverage may be inadequate or even non-existent.

Automated testing frameworks [2, 4, 5, 7, 10, 12] and formal model checking [6, 8, 11] have also been applied to increase testing capabilities and software reliability. While these approaches can be effective in their specific goals, many focus on the discovery and elimination of bugs in software packages. Instead, we assume that these bugs will exist in real-world software despite these existing tools and therefore patching must be done with caution and managed in an efficient and effective manner to avoid negatively impacting software reliability. PatchAdvisor stresses a practical approach to improve manageability of the patching process. While our approach currently focuses on patches to software code, interesting research has been done in the area of configuration management as well [14].

3 The PatchAdvisor Approach

The primary goal of the PatchAdvisor approach is to determine if the changes made by a patch affect areas of the software that are important to its operation in a real deployment. There is an inherent tension between increased risk of adverse negative effects on the software when choosing to upgrade and the forfeiture of the potential positive effects provided by a patch when choosing not to upgrade.

Our approach is based on the simple observation that patching an area of code that is frequently used during execution is likely to have a greater impact, whether positive or negative, on the operation of the software. On one extreme, patching areas of the software that are not exercised during execution may have little value if it doesn't improve the software's operation, but also has little risk of adverse impact. On the other extreme, patching areas that are core to the software's functionality and heavily used during execution may have positive benefits, but also present a much greater risk of adverse impact. In between these extremes, there are many cases where the potential risks and potential benefits have unique trade-offs. The PatchAdvisor approach aims to strike a balance of trade-offs and allow an administrator to maximize the positive benefits of a patch while minimizing the risk of adverse affects.

In order to determine the areas of an application modified by a patch, the actual areas of code that are executed during the software's normal operation, and the impact that the patch will have, PatchAdvisor employs a combination of static analysis techniques and dynamic execution tracing. Our approach consists of three distinct

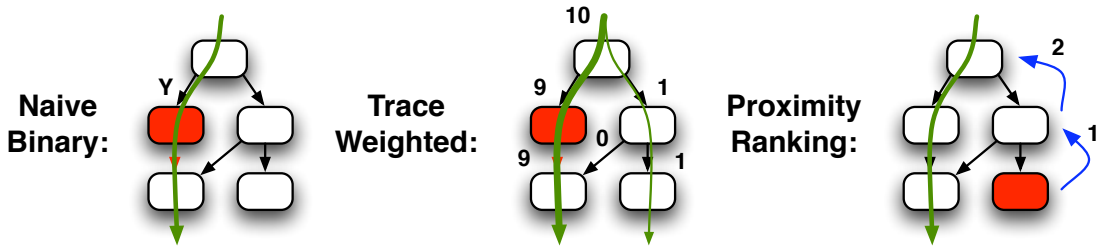

Figure 2: Several examples of our proposed patch impact ranking functions. The red areas represent the modifications of the CFG by the patch and the green lines represent the overlaid execution traces.

stages: (1) a preprocessing stage where the control flow graphs (CFGs) for the software package (both before and after the patch) are generated and compared with each other to identify modified control flow; (2) execution tracing of the application and overlay of the execution traces on the CFG; and (3) the analysis stage where ranking and heuristic functions are applied to infer the impact of the patch. An overview of this approach is pictured in Figure 1. Each of these stages is discussed in further detail in the following subsections.

3.1 CFG Preprocessing

The first stage is generating control flow graphs (CFGs) of the candidate application via static analysis of the application's source code or binary machine code. First, the CFG of the pre-patch version of the application is generated. The patch is then applied to the application and the CFG generation is re-run on the post-patch version of the application. Lastly, the resulting pre-patch and post-patch CFGs are compared to determine what basic blocks of the software were modified by the patch.

3.2 Dynamic Tracing and CFG Overlay

The second stage of PatchAdvisor traces the execution of the candidate application in its normal operating environment in order to get a sense of the execution paths that are exercised. By monitoring branches and jumps in control flow during execution of the application, traces of the execution paths are collected. These collected execution traces are then overlaid on the generated CFGs.

As constant tracing may impose an undesirable performance penalty, execution traces may be collected by sampling periodically depending on needs of the administrator and the deployment environment. Like a debugger, PatchAdvisor can attach to a process to collect an execution trace and then detach, allowing the process to continue running uninterrupted at native speed.

3.3 Impact Analysis and Report

The last stage is responsible for applying functions and heuristics to the modified CFGs overlaid with the execution traces. By determining how the CFG changes interact with the paths of real execution, PatchAdvisor can use these functions to infer how much impact the patch will have on the real-world operation of the software. We briefly describe several of these potential functions, which are also pictured in Figure 2, that can be used to infer patch impact:

- **Naive Binary:** A binary yes/no function can be used to simply say whether or not an execution trace intersects with a portion of the CFG that is modified by the patch.
- **Trace Weighted:** Instead of a simple binary result, the impact of a patch may be ranked by weighting the edges of the CFGs based on how many recorded execution paths traversed that edge. For example, a patch that modifies a core area of a program will likely have numerous execution traces that pass through that core area, resulting in a larger weight and larger risk of adverse impact.
- **Proximity Ranking:** While an execution path may not directly pass through an area of the CFG modified by a patch, it may modify a nearby area. Changing areas of the CFG that are "close" to captured execution traces are arguably more likely to cause an impact on some uncaptured execution trace than areas that are "far" from any traces. The proximity heuristic measures the distance (or number of jumps/branches) from the modified CFG area to the execution trace.

Based on the output of these functions, a patch impact report can be generated and presented to the administrator. The impact report can vary based on the functions used to perform the impact analysis as well as the needs and skills of the administrator. For example, if the trace weighted function is used, the total weight of patch-affected areas can be normalized against the traces

of non-affected areas and assigned an impact score on a human-understandable scale of 1-10 (with 10 being a high impact and 1 being a low impact). Previous upgrades and failures of the particular software package can also be taken into account when determining the score to assign. If an administrator is knowledgeable about the operation of the application and desires more information, an impact report can provide details on functions affected, the set of inputs that reach affected areas, and other operational specifics. With more sophisticated analysis, the report may be able to infer why areas of code were changed and what semantic affect it has on the operation of the application. Such sophisticated analysis is explored further in Section 5.

It is important to note that the integration and development of more sophisticated heuristics into the PatchAdvisor system is a simple process. We envision a hybrid impact function that is composed of several heuristics to generate a final impact report.

4 Implementation and Evaluation

In this section, we describe our initial research prototype implementation of the PatchAdvisor system. We also perform a simple evaluation of its effectiveness in a scenario involving a real-world web application stack.

4.1 Prototype Implementation

We implement our PatchAdvisor prototype on top of the IDA [9] and PaiMei [13] frameworks. IDA is a disassembler and debugger most commonly used in malware analysis and vulnerability research. However, due to its rich functionality and ability to be extended and scripted, the IDA framework is well-suited for a wide range of tasks. IDA's binary disassembler functionality also allow us to apply our PatchAdvisor system to cases where the source code may not be available for the application being patched. PaiMei is a reverse engineering framework offering abstractions that allow powerful reverse code engineering (RCE) tools to be developed on top of IDA. The rapid development and extensibility offered by these frameworks allows to us to determine the feasibility of our approach in an efficient manner.

Our prototype implementation consists of three primary components. The first leverages core functionality of IDA to generate control flow graphs for the candidate application and a modified version of PaiMeidiff tool to determine differences between the pre-patch and post-patch versions of the application. The second component, which records execution traces from live runs of the application by tracking branches and jumps in the runtime control flow, is implemented as a Python application that interfaces through the PaiMei framework. Lastly, the third component determines the intersection of the execution traces with the modified areas of the patched software and implements the impact ranking functions. While our implementation is not yet fully automated in its prototype form (for example, it takes the pre-patch and post-patch binaries as input instead of automating the patch application and compilation process), it is sufficient to demonstrate our approach.

4.2 Preliminary Evaluation

For our evaluation, we demonstrate the PatchAdvisor system using a popular web application stack. Web applications tend to be interesting to investigate as they commonly rely on a number of layers of functionality (e.g., the OS, HTTP server, script interpreter, database layer). While these layers are designed to operate independent of each other, small changes at any layer can often have a significant impact on the entire application.

We performed our experiments on a TurboGears deployment, a popular Python-based web framework. In particular, we employed the SQLObject database layer which communicates with the backend PostgreSQL database using the Psycopg2 library. To test our PatchAdvisor implementation, we analyzed a particular upgrade available for the Psycopg2 library. The upgrade from Psycopg2 2.0.2 to 2.0.3 seemed like a simple upgrade. It was only a revision number version increment and there were a handful of entries in the ChangeLog. However, in one of the bug fixes included in 2.0.3, a programmer mistake caused a NULL pointer dereference to occur in a fairly common code path. In particular, the NULL dereference was introduced into the `typecast_FLOAT_cast` function which converts PostgreSQL FLOAT-typed values from a database query result into a Python float type. Version 2.0.2 of the `typecast_FLOAT_cast` function had a total size of 136 bytes made up of 6 basic blocks and 46 instructions while the version in 2.0.3 had a total size of 107 bytes made up of 3 basic blocks and 38 instructions.

As our web application made use of the PostgreSQL FLOAT type in several columns of the database schema and was accessed in common SQL SELECT statements, the control flow of the program frequently exercised the area of the code modified by the patch. The warnings provided by PatchAdvisor may have prompted an administrator to be more cautious upgrading a hot code path, take a second look at the upgrade, or at least do further testing on the application inputs that intersect with the modified portions of the software. Interestingly enough, the Psycopg2 library does ship with a built-in test suite, but the test cases did not have sufficient coverage to expose this bug before the software was released.

5 Research Directions and Challenges

While our prototype PatchAdvisor implementation can provide useful information regarding the impact of software patches, we believe that there is still much progress that can be made. This area of research presents additional difficult, yet interesting and relevant, problems that are worthy of further investigation:

- **Improved Ranking and Heuristics:** While our initial work has presented several ranking and heuristic functions to determine patch impact, we plan to develop a more sophisticated engine to improve our results. Investigation into additional real-world failures will allow us to tune our engine to more appropriately to match practical considerations.

- **Application-Specific Knowledge:** Using knowledge specific to a specific application, a class of applications, or a particular deployment model may assist in patch analysis. For example, web applications often have well-defined request-response access patterns that may be leveraged to better understand the structure and execution of the application and the effect of a patch.

- **Patch Classification:** It may be possible to automatically classify whether a particular patch is intended to address performance issues, fix semantic application bugs, close security holes, or simply implement new functionality. For example, administrators may wish to only apply security fixes while avoiding new functionality which may not be a native option with some software products.

- **Patch Splicing:** Being able to selectively splice out and apply individual changes from a composite upgrade or from revision control repositories while ignoring others may be useful to an administrator who wishes to include low risk patches yet avoid high risk ones. Determining individual dependencies within a composite patch is a challenging problem, especially if source code is not available.

- **Hands-Free Upgrades:** Automatically and accurately analyzing patches so that no human operator intervention is necessary to perform optimal software upgrades is the holy grail of this problem domain. While this is a lofty goal that is likely impossible to achieve in practice, we can certainly strive towards this goal and provide deeper and more actionable information to assist administrators.

6 Conclusions

In this paper, we have proposed a novel approach to analyze software patches and infer their potential impact on the reliability, performance, and security of software. Providing such information to administrators allows for more intelligent and informed decisions to be made during the patch management process. Our proposed system, PatchAdvisor, combines static analysis of the candidate software and patch, dynamic tracing of real execution paths, and ranking heuristics to automatically infer patch impact. While our initial prototype of PatchAdvisor shows promise in real-world examples, much more sophisticated analysis is possible to significantly improve its results. We believe the research area of patch analysis presents many interesting and difficult problems worthy of further investigation in the future.

References

[1] A. Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In *International Conference on Software Engineering*, pages 85–103. IEEE Computer Society Washington, DC, USA, 2007.

[2] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, December 2008.

[3] George Candea. Towards Quantifying System Manageability. In *Proceedings of the 4th Workshop on Hot Topics in System Dependability*, December 2008.

[4] C. Csallner and Y. Smaragdakis. DSD-Crasher: a hybrid analysis tool for bug finding. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 245–254. ACM New York, NY, USA, 2006.

[5] S.H. Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification & Reliability*, 11(2):97–111, 2001.

[6] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–54, 2007.

[7] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, pages 213–223, 2005.

[8] A. Groce, G. Holzmann, and R. Joshi. Randomized Differential Testing as a Prelude to Formal Verification. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 621–631, 2007.

[9] Hex-Rays SA. IDA Pro Disassembler. `http://www.hex-rays.com/idapro/`, 2008.

[10] R. Majumdar and K. Sen. Hybrid Concolic Testing. In *Proceedings of the 29th International Conference on Software Engineering*, pages 416–426. IEEE Computer Society Washington, DC, USA, 2007.

[11] L. Mariani and M. Pezzè. A Technique for Verifying Component-Based Software. *Electronic Notes in Theoretical Computer Science*, 116(1):17–30, 2005.

[12] C. Pacheco, S.K. Lahiri, M.D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84. IEEE Computer Society Washington, DC, USA, 2007.

[13] Pedram Amini. PaiMei. `http://pedram.redhive.com/PaiMei/`, 2007.

[14] Y.Y. Su, M. Attariyan, and J. Flinn. AutoBash: improving configuration management with operating system causality analysis. In *ACM SIGOPS*, pages 237–250, 2007.