CRANE: Failure Prediction, Change Analysis and Test Prioritization in Practice - Experiences from Windows

Jacek Czerwonka[1], Rajiv Das[1], Nachiappan Nagappan[2], Alex Tarvo[1], Alex Teterev[1]
[1] *Windows Sustained Engineering, Core Operating Systems Division, Microsoft Corporation*
[2] *Microsoft Research*
{ jacekcz, rajivdas, nachin, alexta, alextet } @ microsoft.com

Abstract **- Building large software systems is difficult. Maintaining large systems is equally hard. Making post-release changes requires not only thorough understanding of the architecture of a software component about to be changed but also its dependencies and interactions with other components in the system. Testing such changes in reasonable time and at a reasonable cost is a difficult problem as infinitely many test cases can be executed for any modification. It is important to obtain a risk assessment of impact of such post-release change fixes. Further, testing of such changes is complicated by the fact that they are applicable to hundreds of millions of users, even the smallest mistakes can translate to a very costly failure and re-work. There has been significant amount of research in the software engineering community on failure prediction, change analysis and test prioritization. Unfortunately, there is little evidence on the use of these techniques in day-to-day software development in industry. In this paper, we present our experiences with CRANE: a failure prediction, change risk analysis and test prioritization system at Microsoft Corporation that leverages existing research for the development and maintenance of Windows Vista. We describe the design of CRANE, validation of its usefulness and effectiveness in practice and our learnings to help enable other organizations to implement similar tools and practices in their environment.**

I. INTRODUCTION

Software maintenance is a set of activities associated with changes to software after it has been delivered to end-users. The IEEE Standard for Software Maintenance [8] defines three main types of maintenance activities:

1. **Corrective** maintenance: reactive modification of a software product to correct discovered faults. This category also includes **emergency** maintenance defined as unscheduled corrective maintenance performed to keep a system operational.
2. **Adaptive** maintenance: modification performed to keep a computer program usable in changing environment.
3. **Perfective** maintenance: modification to improve some aspect of quality, such as performance or reliability

The amount of effort going into each of these categories varies depending on the nature of the software considered, its intended purpose, size and characteristics of its current user base. However, the software maintenance phase exhibits attributes that are common across different software products:

1. Software maintenance is expensive. It is generally accepted that the maintenance phase consumes the majority of resources required to take a software product throughout its lifecycle, from inception until end-of-life. The total cost of maintenance is estimated to comprise 50% or more of total life-cycle costs. [24]
2. Maintenance work is often done by people who had not created the system. Unless the effective lifetime of a software product is relatively short, it is expected that the original designers, developers and testers are no longer involved in changes to the product. Reverse engineering might be necessary in the absence of good documentation and institutional knowledge.
3. The size of the maintenance engineering team is typically much smaller than the size of the development engineering team required to create the product in the first place.
4. Changes in deployed software carry a high risk due to possibility of introducing unwanted behavior (a *software regression*). Customer's tolerance to such breaking changes is low.
5. Frequently, time for creating and verifying a fix is constrained (E.g. security-related fixes). Verification and validation of such fixes needs to be done efficiently.

Due to these characteristics, testing of changes in deployed software is essentially a different kind of activity from software testing done before release. Even though software maintenance testing deals with fewer changes, it typically needs to happen in very limited time and often with limited resources. On the other hand, increased risk and cost of making mistakes might warrant expanded test scope. These two competing forces create a challenging environment. One way of addressing this issue is to ensure appropriate information is collected and used to guide and focus test efforts. Data should allow engineers to understand system-wide implications of the change and risks involved.

To address this problem the Windows Serviceability team, the team that does software maintenance for the Windows operating system family, uses a set of software metrics that are collected from static analysis of source code, dynamic analysis of tests running on the system, and field data. All these data sources are together used to predict risk and impact of a change and to guide re-testing of modified code by answering the following questions:

- How risky is the fix we are about to make?
- Which parts of the change are the riskiest?
- Which subset of existing test cases should be executed to maximize the chances of finding defects in the changed code?
- Which parts of the change will not be covered by existing tests and need new tests?
- What dependent parts of the system need to be retested?
- For code that exposes a public interface, which consumers of the APIs should be verified?

In this paper, we propose the use of carefully selected data and processes to perform a comprehensive analysis of risks of fixes, surfacing risk mitigation techniques existing in the system, indicating a need for new risk mitigations. We describe functionality and architecture of an existing system called CRANE that realizes the above goals. Lastly, we provide information on the effectiveness of the system and describe how it is being used as part of Windows servicing workflow.

The remainder of this paper is organized as follows. In section 2, we describe Windows servicing landscape and servicing challenges related to Windows. Section 4 presents the operational details of CRANE. Sections 5 and 6 present the usage of CRANE in failure prediction, risk analysis and test prioritization. In section 7, we conclude and outline areas of future work on CRANE.

II. WINDOWS SERVICING LANDSCAPE

A. Context

The Windows Serviceability team is a several hundred person organization distributed across different countries that is responsible for software maintenance of the Windows operating system family. The Windows Serviceability team at any point in time supports multiple versions of the Windows operating system. Current support includes Windows XP, Server 2003, Vista, Windows Server 2008, Windows 7 etc. End-customers receiving updates to Windows operating system can be broadly divided onto four groups:

1. **Home users** predominantly use the client version of the operating system like Windows Vista, Windows XP, or Windows Media Center and want to keep their PCs in good working condition.
2. **Small and medium-size businesses**, as well as **Enterprises** that use both Windows server and client releases and want to protect their intellectual property, avoid work stoppage, keep their maintenance costs low and want their investments in IT infrastructure working reliably.
3. **Original equipment manufacturers (OEMs)** and **Independent hardware vendors (IHVs)** produce PCs and devices that run with Windows and want the operating system to support the newest hardware so that their customers can obtain the best possible experience.
4. **Independent software vendors (ISVs)** produce applications for Windows and need compatibility between different Windows versions and support for all the application features.

During the post-release phase, any of these customer groups might be the primary target of a fix. All of our customers however expect two things when a Windows fix is requested - quality, defined as seamless integration and lack of change in behavior (if a change in behavior was intended, it should be backwards compatible, i.e. existing Windows applications should continue to function the same way as before), and reasonable turnaround time between reporting a problem and having a fix ready for deployment.

When a new Windows OS version is released publically to customers (called release to manufacturing or "RTM"), the servicing team assumes ownership of all the source code used to build that version of Windows and will use it to produce *hotfixes*. A Windows hotfix is a packaged set of binaries that fix problems, address issues, improve performance, and fix security bugs etc. in the released version of the OS. When installed, these new binaries usually replace their previous versions.

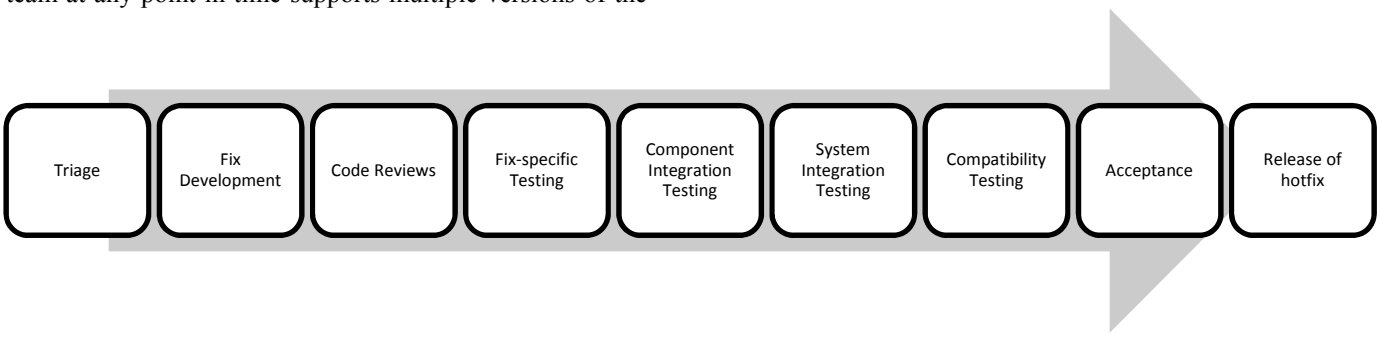

Figure 1: Hotfix engineering process

One feature of the Windows servicing process is a cumulative nature of post-RTM code changes. When a fix to binary FOO.EXE is made for the first time, the resulting package will contain only change $\mathbf{a_1}$. When a later change $\mathbf{a_2}$ is made to the same binary, that new hotfix will contain binary FOO.EXE with both changes $\mathbf{a_1}$ and $\mathbf{a_2}$. This approach simplifies code maintenance but as time goes on and more fixes are implemented, the likelihood of the resulting hotfix containing more than just one change increases. Consequently, if any of the changes are faulty and the failure is not discovered early, quality of all subsequent hotfixes will be negatively affected. In addition to this, all hotfixes should be synchronized with the various Windows versions across multiple releases like XP, Server 2003 and Vista, and also across different process architectures like x86 and x64 bit operating systems. Additionally the fixes are also verified across 30+ language packs like English (US), German and Russian. This discussion indicates the complexity of the Windows servicing landscape.

B. Hotfix engineering process

All problem reports related to Windows go through one of the product support channels. A small portion of these support cases turns out to be true code defects (corrective maintenance) or requests to change behavior (adaptive maintenance). The Windows Serviceability team assumes responsibility for implementing necessary changes in both cases and starts the hotfix engineering process (Figure 1). Each hotfix request begins with *triage* that involves representatives of business management, development, testing, product support, and sometimes the customer as well in a brainstorming session. During the triage, stakeholders evaluate available and feasible workarounds, potential methods of fixing the problem, as well as risks and efforts required from development and testing. As a result, they conclude if the change in the Windows code is necessary or a workaround can be used. Assuming the fix is approved by all parties (Fixes might not turn into a code change if the customer accepts a feasible workaround.); developers then implement the fix and get it code reviewed. At the same time, test engineers prepare and carry out their test plans for testing of the hotfix.

Hotfix testing typically consists of the following phases:

1. **Fix-specific testing or unit testing**: This phase is intended to verify that the fix itself, if it in fact corrects the intended problem and does not produce any regressions.
2. **Component integration testing:** Windows is componentized and all binaries have a place within some component area; interfaces between these areas serve as boundaries of component testing. The objective of this stage is to find and remove any regression in behavior within the component itself as well as at each of its interfaces.
3. **System integration testing**: The changed component might have dependencies from other components that use it through its interfaces. To ensure that the change does not cause regression in other parts of the system, each such dependent component might require retesting, at least in places where it calls into changed code.
4. **Compatibility testing**: Any change with a potential to impact third-party code running as part of the system must be tested against affected software (applications and drivers are two typical examples of third-party code running on Windows).

When sufficient confidence in the quality of the fix is gained, it is sent to the customer for final acceptance. Upon approval, the hotfix is released. As available hotfixes accumulate over time, deployment can become a time-consuming affair. Therefore, periodically all fixes are rolled up to create a Service Pack. Service Packs are well tested since time constraints are not as strict as for individual hotfixes.

III. RELATED WORK

There has been significant research in the software engineering community on defect prediction. This subsection is intended as a broad introduction to a few large industrial empirical studies on failure/defect prediction.

Graves et al. [5] predict fault incidences using software change history based on a weighted time damp model using the sum of contributions from all changes to a module, where large and/or recent changes contribute the most to fault potential [5]. Mockus et al. [9] predict the customer perceived quality using logistic regression for a commercial telecommunications system (of seven million LOC) by utilizing external factors like hardware configurations, software platforms, amount of usage and deployment issues. They observed an increase in probability of failure of twenty times by accounting for such measures in their predictions. Khoshgoftaar et al. [17] studied two consecutive releases of a large legacy system (containing over 38,000 procedures in 171 modules) for telecommunications. Discriminant analysis identified fault-prone modules based on 16 static software product metrics. Their model when used on the second release showed 21.7% type I,19.1% II and 21.0% overall misclassification rates for identifying fault-prone modules.

Schröter et al. [20] showed that import dependencies can predict defects. They proposed an alternate way of predicting failures for Java classes. Rather than looking at the complexity of a class, they looked exclusively at the components that a class uses. For Eclipse, the open source IDE they found that using compiler packages results in a significantly higher failure-proneness (71%) than using GUI packages (14%). Ostrand et al. [17] use code measures in a negative binomial regression equation to predict the number of faults in a multiple release software system (size of the last release was 538 KLOC). The top

20% of the files so identified as fault-prone for fifteen consecutive releases representing four years of field usage and contained between 71% and 93% (average 84%) of the total faults in each release [6]. Biyani and Santhanam [1] show for four industrial systems at IBM there is a very strong relationship between development defects per module and field defects per module. This allows building of prediction models based on development defects to identify field defects. Additionally structural object-orientation (OO) measurements, such as those in the CK OO metric suite [2], have been used to evaluate and predict fault-proneness [11]. There has been an extensive body of research in the failure prediction and test prioritization [7, 19] communities. We intend for this section merely to set up the context for the research presented in this paper[3]. A systematic review related work in defect prediction can be found in [4].

IV. CRANE TOOLSET

During triage, decisions will have to be made on the *proposed* fix. Risk assessment of the proposed fix is needed to plan for potential mitigations. The ***Triage Report*** allows users to look at the various data points for the part of Windows about to be changed, viz.:

1. Vital metrics for the candidate executable and the procedures.
2. History of changes done to this part of code.
3. Data on quality of the test process for the area being changed.
4. Initial approximation of the list of system components to be re-tested.
5. Initial approximation of risk of regressing existing behavior.
6. Available test cases that can exercise the proposed change.

Once the fix is implemented, we collect and report additional data points using the ***Hotfix Report***. The goal is to surface risks and inform about existing and potential quality assurance activities needed to mitigate the risks:

1. Detailed package content.
2. Quality of the test process for the changed area.
3. Existing tests likely to uncover defects in the changed code.
4. Changed lines of code for which tests do not exist.
5. Dependent components.
6. Known third-party software affected by the fix.
7. Final approximation of risk of regression for the fix.

When the fix is finally included in an integration test pass, an ***Integration Report*** is generated. It provides a summary of the various data points like applicable tests and impacted third-party applications over all fixes. Any fix being analyzed might be only one of many fixes tested in the same pass so it is important to prioritize fixes to focus the testers' attention. Secondly, testers involved in the integration pass collectively are responsible for all parts of the operating system; for many of the testers their owned areas do not change when testing commences. Thirdly, depending on the number and size of changes, data provided by CRANE might need to be filtered for the user or her team. While we focus on the most fundamental report delivered by CRANE – the Hotfix report, the description will hold good for most of the Triage report and the Integration report since they implement similar concepts. Any differences for these reports will be explicitly pointed out.

CRANE processing can be divided into three main stages:

1. **Offline Data Collection**: Windows development process data is mined from various sources and cached.
2. **Fix Analysis**: automated and manually initiated processing of code changes against the previously collected data and interpretation of results.
3. **Presentation and visualization** of the analysis results

CRANE's data collection tools gather data from different sources pertaining to various aspects of Windows development and testing process. The bottom layer of Figure 2 shows the data collection process. Examples of data collected include binary dependencies, binary code churn, regression history, details of fixes, binary metrics, code coverage data. Collected binary history information is used to build regression prediction models. These failure prediction models leverage research performed at Microsoft Research on failure-prediction using various metrics [12, 13, 15, 16]. For example Table 1 shows the various precision and recall accuracies for predicting Failure-proneness in Windows Vista using several process and product metrics.

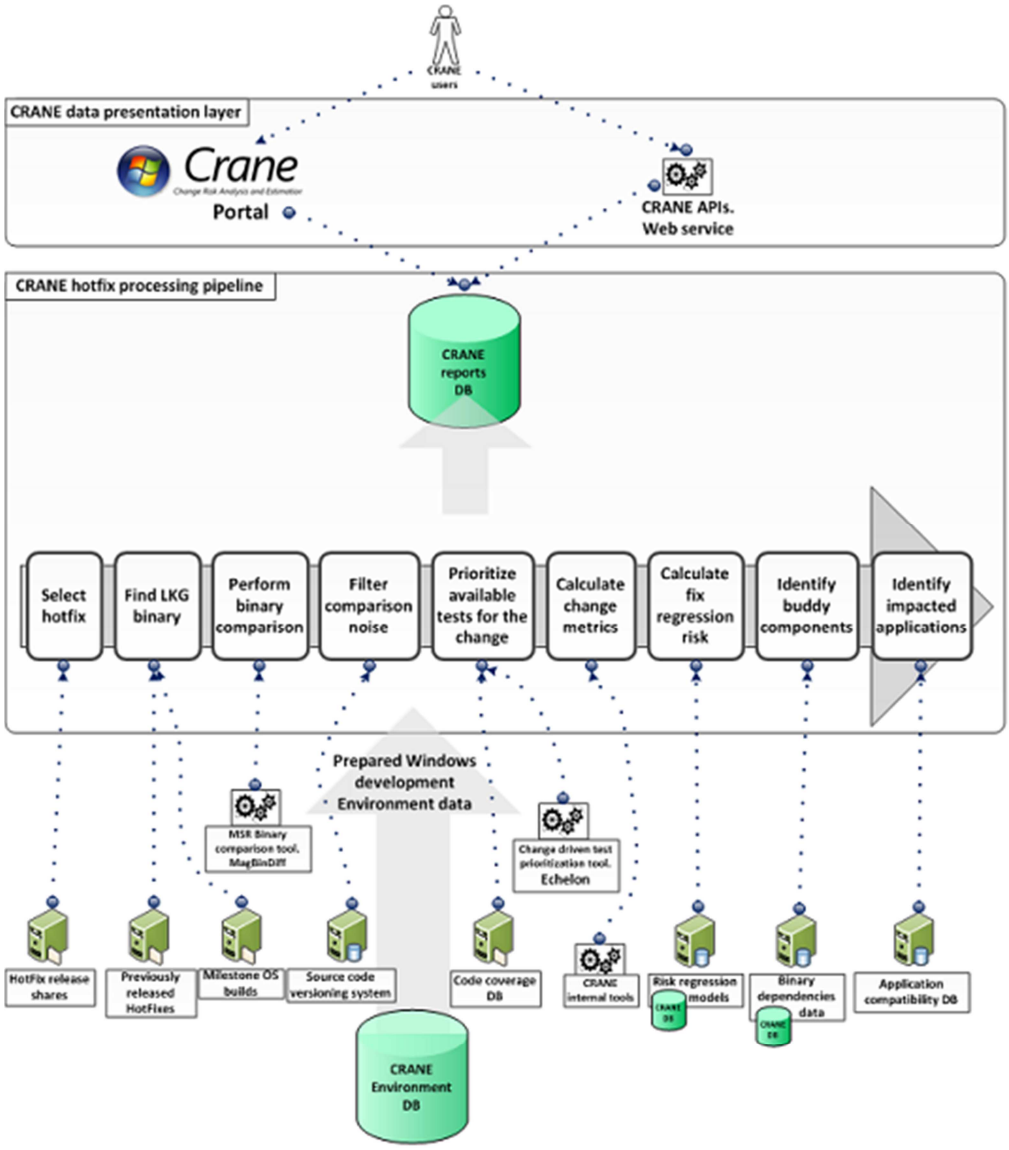

Figure 2: Hotfix analysis process

Table 1: Overall model accuracy using different software measures [16]

Model	Precision	Recall
Organizational Structure	86.2%	84.0%
Code Churn	78.6%	79.9%
Code Complexity	79.3%	66.0%
Dependencies	74.4%	69.9%
Code Coverage	83.8%	54.4%
Pre-Release Bugs	73.8%	62.9%

Once all relevant data is collected and preprocessed, CRANE is ready to accept requests for processing Windows code changes. The middle layer of Figure 2 shows the steps involved in processing.

1. CRANE automatically scans the hotfix release share and initiates analysis of a new hotfix.
2. For each binary in the hotfix, the *Last Known Good (LKG) version* (most recently released or the most recently well-tested) is retrieved from the binary store
3. Code churn from the LKG version is determined using BMAT differencing algorithm [10].
4. Noise in BMAT output (changes not actually made) is filtered out by crosschecking the diff-output with source code differences from the code versioning system. The cleaned outcome represents the actual code changes.
5. For these changes prioritization of available tests is done using Echelon [22].

6. Relevant change metrics are retrieved and regression risk is calculated using statistical models.
7. Impacted components and impacted applications are identified using dependency data and application profiles mined previously.
8. The final report is saved to a database and can be either viewed through the CRANE portal or integrated in other tools through CRANE APIs.

V. FAILURE PREDICTION

The first step to evaluating impact and risk of a change is to understand the exact extent of changes. This can be analyzed by looking at historical data for the affected binaries and on the actual code changes.

Figure 3 shows the various historical and basic data points for binaries that are made available for end-users including complexity, failure-proneness risk (part 1), current coverage (part 2), churn history (part 3), dependent components (part 4), impacted components (part 5) available tests (part 6), and overall binary regression risk (part 1). Churn details includes all changes done to the executable along with details like type of a fix, release date, downloads by users and whether the fix needed any re-work after release (part 7). The last two data points can indicate the level of confidence we can have in the quality of previous fixes. All the historical data gives a broad idea about the nature of failures in the affected binaries.

Once the fix is coded up, CRANE can compare the new binary with the LKG version and determine the actual changes. In many cases, source code might be compiled into multiple executables and the changed executables might force inclusion of additional dependent binaries in the package as well. Moreover, even though testers are most often interested in perusing the last code modification, sometimes they also want to know the extent of changes done since the last broadly distributed release of the executable. For example, they might be interested in all the changes happened in the binary since the last Service Pack. The reason is that broad releases have typically gone through a very extensive and rigorous test process and, more importantly, have already been deployed in the field and their level of quality is often well understood. CRANE shows the number and the extent of changes done since the LKG version of each binary. Figure 3 only shows one branch in detail. In the tool, however we show details for each baseline code branch affected by the fix (figure 3, part 8). With this information, engineers are able to look at the same change in multiple contexts and decide if full retesting in each branch is necessary. Often similarities between code changes can be exploited to shorten test execution time. When multiple fixes are tested at the same time, it is also useful to rank fixes by their risk of regression to help test engineers concentrate on most risky changes.

We have been able to successfully predict the risk of regression and achieve a high degree of accuracy allowing us to distinguish clearly between high and low risk fixes. This in turn allows us to target a subset of fixes that is most likely to contain defects. What specific risk mitigation actions should happen for such fixes is the subject of additional processes, namely test selection, identification of impacted components and impacted third-party code and identification of test gaps.

Failure-proneness is the probability that a particular software element (such as a binary) will fail in the operation in the field [14]. As discussed earlier attributes typically considered good predictors of failure proneness can be module size and complexity, past churn in the module and metrics pertaining to the organization producing the software being evaluated. We re-purposed methods described in research to come up with models using *fix-specific* metrics in an attempt to estimate the likelihood of the fix being defective (all the prior work was on the probability of a file, binary, component being defective). Using logistic regression, we create hotfix failure prediction (called regression risk to clearly highlight the differences in these probabilities to determine if a fix itself will be defective) formulas tailored for our product and the servicing process so as to distinguish accurately between high and low risk fixes at different stages of the process (Figure 3, part 1). Fixes were then classified into one of five buckets: very high risk, high, average, low and very low risk based on empirically assigned thresholds in the [0, 1] range. We chose this method over displaying the actual probability values so that engineers can easily grasp the difference between values.

The Fix Regression Proneness (FRP) model is used to determine the risk of failure in the context of a specific fix. Because we need the flexibility of making decisions early in the process and the ability to refine our risk assessment as data becomes available we have developed two variations of FRP:

- **Simplified FRP, used before a fix is available,** considers only data points, which are easy to obtain without implementing the fix. Simplified FRP requires a user to provide information about which binaries the change will affect, the size of the change in terms of the number of modified source files, and whether the modification is a design change. The rest of the predictors (historical complexity, historical hotfixes) are automatically extracted for the appropriate binary (or binaries).
- **Detailed FRP, used after the fix is implemented,** has a higher accuracy but relies on predictors like number of lines of code deleted by the change, that are obtainable only after the fix is implemented. CRANE uses the Detailed FRP when processing final fixes and this data point is available as part of the hotfix report. See Figure 3, part 1 where the failure probability is based on churn, complexity, prior hotfixes, churn and dependencies.

To evaluate empirically the efficacy of our results we analyzed all fixes made to Windows for one year. Table 2

represents the results of this evaluation. 46.2 % fixes identified by CRANE as very high risk had failed in the field. 17% if the fixes identified by CRANE as high risk failed as well in the field. These numbers as desired are in decreasing magnitude as we do not want all fixes identified as low risk, failing and vice-versa - none of the fixes identified as high risk failing (as significant amount of effort is spent on testing high-risk fixes). These results were the crucial results that showed the several hundred person strong WinSE team the utility of CRANE and lead to increased trust and higher usage.

Table 2: Retrospective evaluation of CRANE

Predicted risk	Very High	High	Average	Low	Very Low
Efficacy	46.2%	17%	4.4%	3.1%	1.56%

A point to be noted is that these are not accuracy/precision/recall values (hence do not sum to 100%). The primary concern for the Windows team is to gain confidence in the system. The way these numbers are interpreted by the teams, when a fix is identified as very high risk by CRANE there is a 46% chance that this fix will fail in operation.

VI. TEST PRIORITIZATION

The purpose of test selection is to identify a subset of available tests that is most likely to be effective in finding defects in changed code. Effective test selection makes it possible to decrease the total cost of running selected tests while keeping defect-finding effectiveness of the formed subset at the maximum level. To perform test selection, CRANE uses Microsoft's Echelon test prioritization scheme [22]. Echelon analyzes differences between two

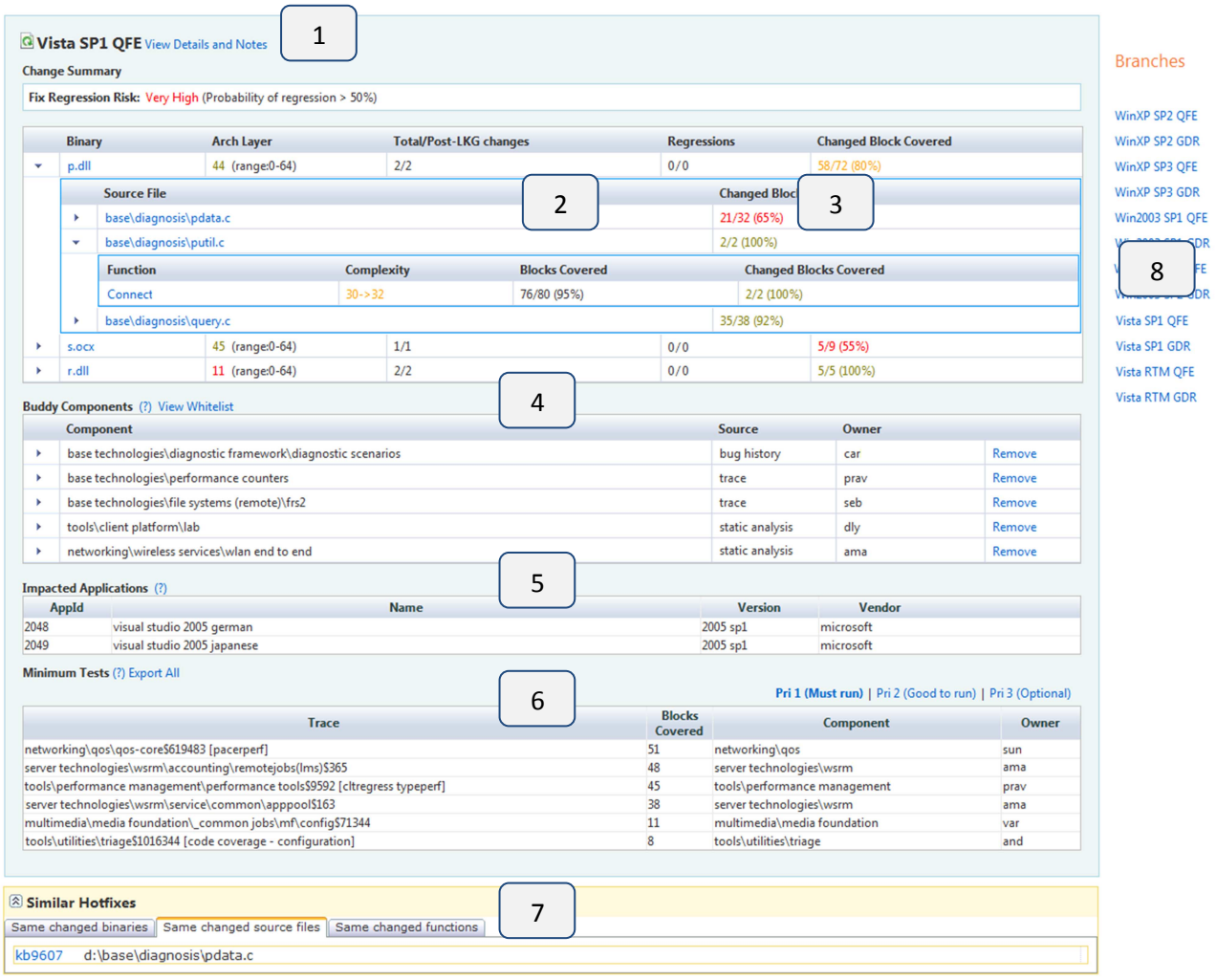

Figure 3: Report for the implemented fix

binaries, at a basic binary block level, and then uses previously archived code coverage information to identify tests that will trigger execution through maximum number of changed binary blocks. Echelon prioritizes the selected tests by "changed blocks covered per test cost unit" ratio. Tests that add more coverage to the changed code per unit of effort will end up at top of the list. At this time, we use Echelon as a test prioritization rather than a pure test selection tool i.e. we do not recommend that only the selected tests be executed on a fix but rather that they are run first. (For example, figure 3, part 6).

For the situations where certain portions of changed code will be identified as not covered through existing tests, these "test gaps" are an important indicator of test cost—ideally all changed code would be executed before the release, therefore new tests need to be defined and run, in our tool a source level view of changes (figure not shown here for reasons of space) represents this information in a form of "green" (covered by existing tests) and "red" (not covered) coloring of all changed lines of code. Our recommendation is that all currently uncovered parts of code have tests developed and executed for them.

Following the prioritization scheme as defined in Echelon[22] has yielded significant cost and efficiency benefits. In addition the documented saving[22] our own studies suggest that the effectiveness of our approach to identify tests which, when executed, identify regressions is between 50 and 63%. To arrive at these numbers we performed three independent case studies. We considered a number of fixes that we knew were defective and asked the following questions: (a) did a test exist in our system, which finds this problem, (b) was this test identified by Echelon and selected for execution for the original fix. For each of the studies we took mutually exclusive fixes. The normalized values are due to the confidential nature of the number of failures in Windows. Table 3 contains the results. The primary purpose of these studies was to gain confidence of the large engineering community of the effectiveness and use of CRANE.

Table 3: Results of studies on Echelon's effectiveness

	Study 1	**Study 2**	**Study 3**
A - Total number of regressed fixes	X	Y	Z
B - Number of fixes with pre-existing tests able to find a problem	0.8 X	0.6Y	Z
C - Number of fixes for which a suitable pre-existing test was identified by Echelon	0.4 X	0.4 Y	0.5 Z
Echelon's effectiveness [C/B]	52%	63%	50%

We plan to enhance further the test selection algorithm by exploring the use of current research in the software engineering community. In the future we also plan to use a more precise definition of test cost comprehensively describing effort needed to configure test environments, execute tests and analyze results and use that to further increase the efficiency of test selection.

VII. LESSONS LEARNED

Windows due to its size, complexity, diverse set of users and role in the PC ecosystem poses a unique maintenance challenge. Expectations for hotfix quality and response time are critical to the continuous success of Windows. The CRANE toolset has been developed to expose engineers to previously hidden information with the purpose of helping them make decisions on the scope of testing required to minimize risks of further problems in changed code. CRANE is an example of a successful toolset from research [10, 14-16, 21, 22] to practice which is built upon prior work in Microsoft Research.

This paper documents the use, engineering trade-offs, deployment and successes of CRANE. Since its deployment, CRANE has been able to help developers and testers find defects by either identifying individual tests or areas of testing likely to uncover problems. Its adoption has increased significantly in recent months by several hundred of users. For context for other industrial organizations, CRANE had an initial investment of six engineers working on it full time plus one researcher from Microsoft Research on loan to Windows to transfer research results into a product. CRANE was used to triangle every bug for Windows Vista SP2, a significant engineering task which was of significant fiscal and technical importance to Microsoft. The total investment was in excess of one Million dollars plus and the total development time from design to operationalize was nine months.

A key take-away during design and development of CRANE was to ensure its accuracy and effectiveness without sacrificing simplicity and usability. In the process, we also came to realize the following underlying principles that are of practical importance in creating data mining tools like ours:

Data should be simple to understand, empirical, insightful. Users need to understand the connection between a given metric and the outcome (preventing regressions in our case). Metrics should provide information that would otherwise be hidden.

Data needs to be project and context specific. The choice of metrics is determined by the project at hand. Even metrics that can be applied universally will have project-specific thresholds above which risk is substantially larger. Statistical analysis of data helps determine these thresholds.

Metrics should be non-redundant. A few carefully chosen data points are easier to use than many numbers. Each data point should add a substantial amount of new information.

Information should be actionable. Metrics should be interpreted and users need to understand how to act based on the data presented. Some apriori assumptions are neces-

sary (i.e. “if you see complexity >= 50 be concerned”) but some of this knowledge can only be accumulated over time as our tools and data points are used.

Have at least one statistical expert in the team. This helps significantly during the model development and deployment phase. Additionally user engineers always had questions on the implications and meanings of the FRP values. A statistical expert was very helpful for the development of CRANE.

Our future work is targeted towards the following areas:

Improved performance of metric collection. In order to roll out CRANE to other product divisions within Microsoft we plan to improve the performance and speed for collecting code metrics information. This involves building robust parsers, investment in engineering efficiency and interconnection between various data sources.

Ensuring validity in a changing development system. Development process is a constantly evolving social system. With our risk prediction work completed, we are able to make better decisions at all stages of the hotfix process. As a result, we expect to see modifications to the process driven by people taking action on data they observe which in turn might affect what predictors we use for calculating risk and how strongly they correlate with defects. We expect we will need to recalculate our models every few months to keep up with these changes.

Efficiency of recommendations. Some of the current recommendations make simplifying assumptions, which might affect efficiency of test execution. For example, a more precise definition of test execution cost will be useful in improving efficiency of test selection.

Effectiveness of recommendations. Change driven test prioritization in combination with recommendations on impacted components and applications has already proven effective in detecting regressions. We plan to experiment with various modifications to the schemes described above in an attempt to increase the effectiveness of our recommendations.

ACKNOWLEDGEMENTS

This work would have not been possible without the support of Bharat Shyam, General Manager of Windows SE, Wael Bahaa-El-Din, Technical fellow, Windows Core Operating Systems Division who funded the headcount staffing and resources in the team to build CRANE; Thomas Ball and Jim Larus from MSR who allowed Nachiappan to be on loan to the Windows team for the fiscal year 2007.

REFERENCES

[1] S. Biyani, Santhanam, P., "Exploring defect data from development and customer usage on software modules over multiple releases", Proceedings of International Symposium on Software Reliability Engineering, pp. 316-320, 1998.

[2] L. C. Briand, J. Wuest, S. Ikonomovski, and H. Lounis, "Investigating quality factors in object-oriented designs: an industrial case study", Proceedings of International Conference on Software Engineering, pp. 345-354, 1999.

[3] C. Catal, Diri, B., "A Systematic Review of Software Fault Prediction Studies", *Expert Systems with Applications*, 36(6), pp. 7346-7354, 2009.

[4] R. Das, Czerwonka, J., Nagappan, N, "Finding Dependencies from Defect History", Proceedings of Industrial Track, International Symposium on Software Reliability Engineering, 2009.

[5] T. L. Graves, Karr, A.F., Marron, J.S., Siy, H., "Predicting Fault Incidence Using Software Change History", *IEEE Transactions in Software Engineering*, 26(7), pp. 653 - 661, 2000.

[6] T. Gyimothy, Ferenc, R., Siket, I., "Empirical validation of object-oriented metrics on open source software for fault prediction", *IEEE Transactions in Software Engineering*, 31(10), pp. 897 - 910, 2005.

[7] M. J. Harrold, Rosenblum, D., Rothermel, G., Weyuker. E., "Empirical Studies of a Prediction Model for Regression Test Selection", *IEEE Transactions in Software Engineering*, 27(3), pp. 248-263, 2001.

[8] IEEE, "IEEE Standard for Software Mainrenance," 1998.

[9] T. M. Khoshgoftaar, Allen, E.B., Goel, N., Nandi, A., McMullan, J., "Detection of Software Modules with high Debug Code Churn in a very large Legacy System", Proceedings of International Symposium on Software Reliability Engineering, pp. 364-371, 1996.

[10] S. McFurling, Pierce, K., Wung, Z., "BMAT – A Binary Matching Tool," Microsoft Research Technical Report MSR-TR-99-83, 1999.

[11] A. Mockus, Zhang, P., Li, P., "Drivers for customer perceived software quality", Proceedings of International Conference on Software Engineering, pp. 225-233, 2005.

[12] N. Nagappan, Ball, T., "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study", Proceedings of International Symposium on Empirical Software Engineering, pp. 364-373, 2007.

[13] N. Nagappan, Ball, T., "Use of Relative Code Churn Measures to Predict System Defect Density", Proceedings of International Conference on Software Engineering, pp. 284-292, 2005.

[14] N. Nagappan, Ball, T., Murphy, B., "Using Historical In-Process and Product Metrics for Early Estimation of Software Failures", Proceedings of International Symposium on Software Reliability Engineering, pp. 62-74, 2006.

[15] N. Nagappan, Ball, T., Zeller, A., "Mining metrics to predict component failures", Proceedings of International Conference on Software Engineering, pp. 452-461, 2006.

[16] N. Nagappan, Murphy, B., Basili, V., "The Influence of Organizational Structure On Software Quality: An Empirical Case Study", Proceedings of International Conference on Software Engineering, pp. 521-530, 2008.

[17] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the Bugs Are", Proceedings of International Symposium on Software Testing and Analysis, pp. 86-96, 2004.

[18] A. Pogdurski, Clarke, L.A., "A Formal Model of Program Dependences and its Implications for Software Testing, Debugging, and Maintenance", *IEEE Transactions in Software Engineering*, 16(9), pp. 965-979, 1990.

[19] G. Rothermel, Harrold, M.J., "A safe, efficient regression test selection technique", *ACM Transactions on Software Engineering and Methodology*, 6(2), pp. 173-210, 1997.

[20] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting Component Failures at Design Time," in *International Symposium on Empirical Software Engineering*, 2006.

[21] A. Srivastava, Edwards, A., Vo, H., "Vulcan: Binary Transformation in a Distributed Environment," Microsoft Research Technical Report MSR-TR-2001-50, 2001.

[22] A. Srivastava, Thiagarajan, J., "Effectively Prioritizing Tests in Development Environment", Proceedings of International Symposium on Software Testing and Analysis, pp. 97-106, 2002.

[23] A. Srivastava, Thiagarajan. J., Schertz, C., "Efficient Integration Testing using Dependency Analysis," Microsoft Research-Technical Report, MSR-TR-2005-94, 2005.

[24] H. V. Vliet, *Software Engineering: Principles and Practices*. West Sussex, England: John Wiley & Sons, 2000.