

# Evaluating Patch Safety: Configuration Sharing for Problem Avoidance

James D. Thornton, Vivien Quéma

25 February 2005

TR-05-2

Portions of this paper are **Copyright 2005, Palo Alto Research Center**

This paper is part of the PARC Technical Report series.

For more information on PARC, please visit our Web site at <http://www.parc.com/>.  
Our address is:

Palo Alto Research Center  
3333 Coyote Hill Road  
Palo Alto, CA 94304 USA

You can contact us via telephone at 650-812-4000.

You can also send e-mail to [info@parc.com](mailto:info@parc.com); it will be forwarded appropriately.

# Evaluating Patch Safety: Configuration Sharing for Problem Avoidance

James D. Thornton

*Palo Alto Research Center, Palo Alto, USA*

Vivien Quéma

*Institut National Polytechnique de Grenoble, INRIA Rhône-Alpes, France*

## Abstract

*Today's complex computing systems with their ever-changing software are very difficult to maintain in proper working order. Flaws and vulnerabilities in released software result in a flood of possibly-buggy patches. Keeping systems patched safely is a major challenge faced by system administrators today. Computing power and Internet connectivity are now being applied to automating problem diagnosis, and we argue that a similar strategy should be used for problem avoidance. We propose a system for global-scale sharing of experience with patches to help users avoid those patches likely to cause problems for them.*

## 1. Introduction

Today's computer systems are powerful and adaptable but they are often hard to manage and maintain. The flexibility of software that makes machines so useful also leads to complexity and instability. Problems are caused by bugs that weren't fixed by authors before release and configuration bugs created on a user's machine over time by installation of conflicting software, mis-setting of parameters and the like. We would like to help users avoid or fix problems of both types.

Given that many of these machines are connected to the Internet, we now have the opportunity to address maintenance problems through sharing information between machines. A number of recent efforts have been using this approach for troubleshooting and finding bugs. For example, Redstone et al. have proposed a vision of large-scale data sharing for problem reporting and diagnosis [7]. The STRIDER system uses state differencing, execution tracing and sharing checkpoints between machines to help users isolate configuration errors in the Windows registry [9]. PeerPressure from the same project uses these techniques in a peer-to-peer overlay network [8]. Finally, Liblit et al. [5] are working on sampling executions in a large user population to identify specific bugs in code.

We have been considering the opportunities for large-scale, anonymous data collection for problem *avoidance* rather than resolution. Our challenge is to identify potential problems before you encounter them, based on the experience of others. Imagine that you could avoid applying a dangerous configuration change to a system based on statistical evidence representing the accumulated experience with that change. We believe that sharing success and failure of varied configuration

options will be an important part of self-healing, self-optimizing, and self-managing systems in the future.

This kind of experience sharing is most natural for configuration changes that large numbers of people are likely to face. Our work focuses on software patches, which amount to significant and risky sets of configuration changes that many people will interact with.

### 1.1. The Patch Evaluation Problem

Deciding whether or not a patch is safe to install is what we refer to as the *patch evaluation* problem. If you manage even one machine, you are probably familiar with the challenge of responding to a continuous flood of new patches. Every patch that comes along has the potential to create new problems while promising to fix or prevent others. One could simply choose not to install patches, but since many now close security holes there are serious risks associated with ignoring them.

Problems with patches have historically occurred frequently enough to lead some to the conclusion expressed by the president of a web design firm who said "In most cases, I'm better off just playing Russian roulette with the hackers until our servers are broken into." [3] As indicated by an attempt to model the risks of patching and not patching [1], information about patch failure is not easy to come by.

The safety of a patch is not a simple binary property because patch experience is not identical across an entire population. Patches with inherent flaws will fail everywhere, but we believe that the more common and serious cases are patches that fail only under particular circumstances. Since there is poorly-understood risk associated with not installing a given patch, a problem avoidance strategy would involve skipping only those security patches that are likely to cause problems in a particular context. Thus we have been looking at sharing configurations as part of sharing patch experience for problem avoidance.

### 1.2. Proposal

We propose a system for global-scale, semi-automated information sharing among patch consumers. Anyone who installs a patch should be able to share their experience with that patch risk-free with little effort. On the other side, anyone evaluating a patch should be able to query the accumulated experience

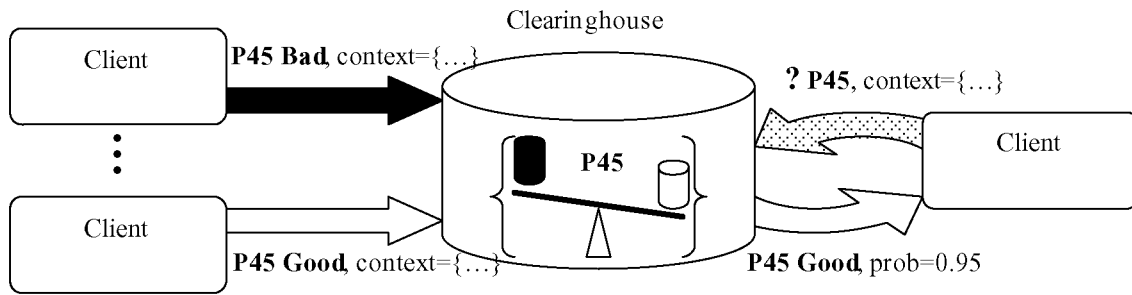


Figure 1: A conceptual block diagram showing interactions for hypothetical patch #45. The top client on the left reports a problem with the patch in its context while the bottom left client reports success. The client on the right queries with its context and the clearinghouse responds that based on all received reports, the client on the right is highly likely to experience success with the patch.

easily and without risk. Ultimately, such a system could support automated evaluations for lightly-managed machines. We will discuss the implementation of this proposal in the remainder of the paper.

## 2. Sharing Patch Experience

A simple way to organize the sharing of patch experience for problem avoidance is illustrated in Figure 1. A client component runs on each participating machine to observe and report on patching activity. A central server, the *clearinghouse*, maintains a database of the collective experience.

In every interaction about a patch, a client provides configuration information as context. When reporting experience, the client describes the configuration on which the patch was applied. When querying, the client describes the state of the machine on which the patch is being evaluated. By running on the participating machine, the client is able to engage the patching process and gather configuration data. It should work in concert with a patch distribution system that decides which patches are applicable and handles installation and de-installation. Avoiding installation of patches that don't apply is a minimal problem avoidance step! The client can also provide an interface to gather additional information from users and leave them in control of the final decisions about what gets installed.

The context information that is shared in experience reports can be used to train statistical classifiers to recognize good and bad contexts for each patch. In the simplest case, suggested by Figure 1, the classifier for a patch would cluster configurations into just two states, good and bad. Note that this simple approach accepts very imprecise and subjective definitions of “good” and “bad”. Since the interactions of patches with a system and its applications can be very complex, we believe that the definition of what constitutes a problem must be left up to the users. Of course, more data about the nature of a problem could be very helpful, particularly indications of perceived severity. We have not yet experimented with classifiers for this problem but we believe that good classifiers can be found when we have a picture of the data.

The clearinghouse must receive and process reports and answer queries for a large number of clients. We have pre-

sented the clearinghouse as a centralized facility for simplicity. That functionality could be distributed for scale and reliability. It could also be provided in a massively distributed way by the clients themselves functioning in a peer-to-peer overlay network in the spirit of [8]. Communication of context can also be much more dynamic and interactive than the sketch in Figure 1 suggests.

The most fundamental questions we face, as for any problem avoidance scenario, are about what to share and when to share it. We will address those questions first, then mention a few other challenges related to encouraging participation.

## 3. Capturing Relevant Context

The data we gather and share must capture features of machines that are useful for classifying states with respect to patch success. In addition, the data must be fast to collect, transfer and process, and must be anonymous. The need for good coverage conflicts with the practical requirement of good performance because the size of configuration state on common systems is very large. For example, the typical number of registry values alone on Windows systems has been cited at over 100 000, though this includes items we might not regard as configuration state [9]. Since effects of a patch can be complex and widespread, we cannot limit data collection to just those features that we would normally consider relevant to a patch, such as the state of components it explicitly depends upon.

The challenge, then, is to select a relevant subset of features of the configuration state, and that is difficult because we don't know in advance what information will be relevant. This is a fundamental challenge for problem avoidance by statistical analysis. In a problem diagnosis context, we can use execution tracing and before/after comparative techniques to filter the configuration space [9]. Without any knowledge of what problem might arise, we must rely on different and less precise strategies.

### 3.1. Exploring features

As a preliminary step, we conducted a survey of machines in routine use to gain some insight into the distribution of config-

uration features. This voluntary, automated survey gathered useful configuration data from 55 Windows machines at PARC. We will suggest some approaches to selecting features, informed partly by results from this survey.

### 3.2. Installed Software

Describing the collection of installed software is an attractive approach to representing the configuration context of patches. It seems likely that more patch problems are dependent on the combination of software that is installed than on detailed configuration settings. We could simply get a list of installed software from the registry, which would be compact while representing a large amount of state. The problem is that the system configuration can drift seriously from this meta-data description, and “unusual” configurations created by different software installations over time are likely to be a significant factor for patches.

In addition to registry meta-data, we could capture installed software state in terms of the states of code files on disk (e.g. .DLL, .EXE). Reading the size, date, and internal version number and checksum (where available) is easy and quick. Actually reading the files to compute a checksum is expensive and to be avoided. The big challenge, however, is that there can be so many code files on a machine. Our survey found an average of 6566 files with one of 18 candidate extensions on the system disk.

We also discovered that most files are rare across a population, as illustrated by Figure 2 which shows the proportion of files in each of 10 buckets by frequency of occurrence. The first bar shows that over 80% of the file names were found on no more than 10% of the machines in our sample. The far right bar shows that fewer than 10% of file names could be found on more than 90% of machines, that is that few names were very common. The heights of all the bars in these graphs sum to 100%. Note that most of the files on a given machine are not rare but there are many uncommon files that could result in very sparse data for training.

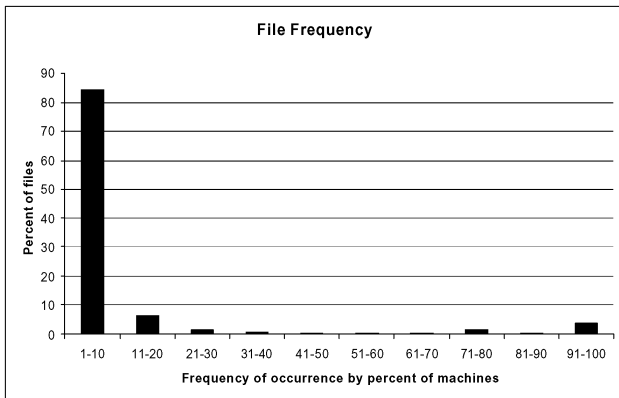


Figure 2: File occurrence frequency across machine population (by file name)

The pattern of Figure 2 showed up everywhere to varying degrees: among files of a single type, in particular system areas of the disk and also in registry data. This reflects the great diversity of software for the Windows platform: it is easy to have some packages unique to each machine in a small sample. We speculate that a global-scale population might show a smaller proportion of very rare items but probably not more very common ones. There is only a limited set of software that enjoys very high penetration across millions of machines. This fact means that a consistent feature set for installed software will be large.

### 3.3. Common Files

Instead of capturing the state of all code files, we could focus on common groups of files likely to be found on many participating machines. This might be sufficient for the most serious patch problems that destabilize the operating system itself or interfere with very popular applications. We wondered whether frequently-occurring files would be very diverse, but our survey showed that even the most common files vary quite a bit even across a small sample. For example, Figure 3 shows the frequency graph among just the file variants for .DLL file names found on every single XP machine in our data set. This data involves only 1697 file names.

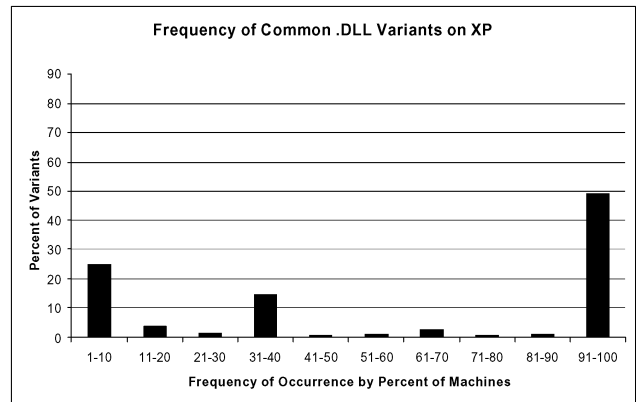


Figure 3: Frequency of variants of .DLL files found on every XP machine

One example of a problem that might be detected using only data about common features is the case of a security patch released in March of 2003 for the Microsoft IIS web server on Windows 2000 [4]. Some customers with certain earlier patches experienced system failures. In this case, a Microsoft bulletin [6] provides enough information to relate the problem to a particular version of the file `ntoskrnl.exe` in the `system32` directory. Unsurprisingly, this file was present on every machine we surveyed, and could easily be part of a collection set of common features. This particular variant of the kernel was reported to have been distributed only with hotfixes for specific problems to premiere customers so it might not have been easy to characterize the problematic conditions. A configuration description with version information for just the

most common files might have distinguished the problematic cases.

### **3.4. Dynamic Strategies**

Allowing interaction between client and clearinghouse offers the opportunity to dynamically adjust what is collected. As more reports are received about a particular patch, the clearinghouse might begin to request additional data from newly reporting clients to refine its classification. Any particular feature set must balance cost and precision, giving a snapshot at a particular resolution. This means that there will be situations that are different but indistinguishable; configurations that have varying results with particular patches but are aliased together at the resolution of the descriptions. If the clearinghouse can alter the collection set over time, it can potentially increase the resolution just where disambiguation is needed. This interactive approach is already used by WindowsUpdate to determine what patches apply to a client machine, apparently to make the process efficient over slow network connections [2].

### **3.5. Timing and Triggers**

We turn now to the question of when to share experience. As noted earlier, we accept a subjective definition of patch failure, so the basic answer about when to share a bad experience is when the user decides that they are having a bad experience. We might give the user an interface to report a problem, though that also requires giving the user some help to figure out whether the problem is related to a patch. We might instead trigger a report from an action that indicates a patch problem, such as de-installation of the patch.

It would be good to receive positive reports about a patch, and these are more difficult to trigger. One approach would be to allow a certain amount of usage of the machine to pass without negative indications and regard that as a tacit endorsement. The classification strategy may well need to accommodate multiple reports from the same machine about the same patch over time.

## **4. Participation Challenges**

Any data sharing arrangement depends upon willingness of people to participate. Convenience and ease of use will be key to widespread participation. This is one major reason why, in the patch evaluation case, we advocate mostly automated experience reporting and coordination with patch installers. Assurance of safety is also important, and to that end we will discuss privacy and security issues. First, however, we want to address a basic timing challenge to participation in a problem avoidance system.

When a particular patch is new, for example, there will be some period of time during which there are few reports at all and as a result little statistical evidence about the safety of the

patch. People must be willing to try patches and contribute reports even when they cannot immediately benefit or else no valuable data will ever be amassed.

At least in the case of patches, we believe that there will always be those who are willing to go first. They may be large institutions with high reliability requirements and the resources to test patches carefully for themselves. These may be motivated to share because they will not always be first to experience a particular problem and cannot test every configuration variation themselves so they do benefit from the system. Other early patch testers might be owners of less-critical machines who can accept higher risk from patches, or perhaps find that preferable to a higher risk of penetration from waiting to patch. A shared patch evaluation system allows early adopters to make their experiences useful for others, which we believe would be attractive to many. Social or possibly even financial rewards could add motivation and preserve quality if carefully designed. Special measures such as trust-based weighting of early reports might improve the value of smaller data sets. Finally, real evidence about the success of patches in the field would motivate and perhaps assist patch producers to fix problems quickly which would presumably benefit all participants.

As time progresses, each participant will become a contributor when their own threshold of evidence is crossed. Lightly-managed machines might patch themselves automatically when some threshold is crossed, which might be more successful at getting large numbers of personal machines patched in a timely fashion than alternative approaches. The key will be to make participation easy and safe.

### **4.1. Privacy**

We propose to ask people to share potentially sensitive information about machine configurations and patch experience, and we must be able to provide strong privacy guarantees. Concerns include exposure of weaknesses to potential attackers, disclosure of personal or proprietary data, and possibly even exposure of internal software usage and practices that might be sensitive for competitive or legal reasons.

We believe that privacy can be adequately protected through strict anonymization implemented with multiple mechanisms. Privacy begins with the client aggressively sanitizing context data to eliminate intrinsically identifying information such as names. Privacy also requires that the clearinghouse preserve no association between the identity of a client and its data. This is easy for a trustworthy clearinghouse implementation to do, and possible to arrange with untrustworthy clearinghouses. There are trade-offs we do not have room to discuss between trust and security mechanisms in the face of a range of possible threats, but we believe that practical solutions are possible.

Eliminating identifiers of people and organizations is a challenge because a number of variations of names and userids may be found on a typical machine. For example, various names appear in the registry to record who installed pieces of software. There is no consistency in naming such values and userids are also found embedded in directory names. We believe that a two-pronged strategy is needed to handle these different variations: (1) eliminate known problem items regardless of what values they have, and (2) eliminate known identifying values regardless of where they occur.

Sanitizing methods such as we have described will always be susceptible to both false positives and false negatives and will need to be adjusted over time as patterns of usage of the registry and filesystem evolve. Some types of data, such as the first names of individuals, may be of little sensitivity when the contributing population is very large. Other types of data, such as variations of organization names may be significant at any scale and difficult to eliminate completely. Our preliminary analysis has certainly not solved the problem of anonymizing, but with careful engineering we believe that an acceptable solution can be produced. As noted earlier, the clearinghouse can also play a role in anonymizing and may be able to partly compensate for limitations in sanitizing individual reports.

#### 4.2. Security

Our proposal also places people in the position of trusting external assessments, making the system attractive to attackers who want to manipulate machines. For example, an attacker preparing to exploit a vulnerability might try to discourage installation of the patch that eliminates that vulnerability by causing that patch to be reported unsafe for many configurations. A more complex attack might involve the introduction of a false patch into the distribution system while simultaneously arranging for it to be reported as safe. Denial of service attacks of various types might be used to delay patching and thus increase the window of vulnerability of participants.

While some attacks may be impossible to prevent in general, we believe that the most serious risks can be addressed by careful implementation and statistical reasoning. The trustworthiness of the client software must first be assured by distribution policies and techniques like code signing. There are then two general attack approaches beyond plain denial of service. First, the attacker may attempt to subvert or masquerade as the clearinghouse. In the case of a centralized clearinghouse design, these attacks can be prevented by having a trustworthy organization operate the clearinghouse and protect it carefully, and by using cryptography to protect clients against spoofing. In the second approach, an attacker may attempt to fool the legitimate clearinghouse by, for example, submitting many bogus reports. A trustworthy clearinghouse can defend against these attacks by detecting repeated submissions, even while anonymity is preserved. Finally, if these approaches are denied to an attacker, they will still be able to submit some number of false reports that appear legitimate, but if those

reports are contradicted by many good reports they will carry little statistical weight and will therefore be ineffective. As with privacy, there are a variety of trade-offs and cases that we do not have space to consider here.

## 5. Summary and Future Work

We have advocated the automated sharing of configuration data for problem avoidance. We have discussed the specific case of avoiding problems caused by software patches and made some preliminary proposals for a global-scale system to share patch experience. We hope to stimulate further thinking about problem avoidance in general within the systems community.

Much more work would be needed to implement a system such as we have proposed for patch problem avoidance. The next major challenge is to obtain significant quantities of data about configurations relevant to patch failure, to support experimentation with classifiers and evaluate hypotheses. Small-scale surveys will not be sufficient. We are thinking about how to obtain useful data.

## 6. References

- [1] Beattie, S.; Arnold, S.; Cowan, C.; Wagle, P.; Wright, C.; and Shostack, A.; "Timing the Application of Security Patches for Optimal Uptime", *Proc. of LISA 2002*, Philadelphia, November 2002, pp. 101-110.
- [2] Hartmann, M.; "Inside Windows-Update", *tecchannel.de*, IDG Interactive, February 2003, <http://www.tecchannel.com/security>
- [3] Lemos, R.; "Microsoft fails Slammer's security test", *CNet News.com*, January 27, 2003.
- [4] Lemos, R.; "Microsoft patch freezes some systems", *CNet News.com*, March 20, 2003, <http://news.cnet.com/2100-1002-993515.html>
- [5] Liblit, B.; Aiken, A.; Zheng, A.; Jordan, M.; "Sampling User Executions for Bug Isolation", *Proc. of RAMSS*, Portland Oregon, May, 2003, pp. 3-6.
- [6] Microsoft Corporation, "Microsoft Security Bulletin MS03-007", *TechNet*, May 30, 2003, <http://www.microsoft.com/technet/treeview/?url=/technet/security/bulletin/MS03-007.asp?tag=nl>
- [7] Redstone, J.; Swift, M. M.; Bershad B. N.; "Using Computers to Diagnose Computer Problems", *Proc. of HotOS IX*, Lihue Hawaii, May 2003, pp. 91-96.
- [8] Wang, H.; Hu, Y.; Yuan, C.; Zhang, Z.; Wang, Y.; "Friends Troubleshooting Network: Towards Privacy-Preserving, Automatic Troubleshooting", in *Proc. of IPTPS'04*, Feb. 2004.
- [9] Wang, Y.; Verbowski, C.; Dunagan, J.; Checn, Y.; Wang, H. J.; Yuan, C.; Zhang, Z.; "STRIDER: A Black-box, State-based, Approach to Change and Configuration Management and Support", *Proc. of LISA 2003*, San Diego California, October 2003, pp. 159-171.