Discovering Correctness Constraints for Self-Management of System Configuration

Emre Kiciman
Yi-Min Wang

March 20, 2004

Technical Report
MSR-TR-2004-22

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

To appear in Proc. IEEE International Conference on Autonomic Computing (ICAC), May 2004.

Discovering Correctness Constraints for Self-Management of System Configuration

Emre Kıcıman and Yi-Min Wang
emrek@cs.stanford.edu, ymwang@microsoft.com

Abstract

Managing the configuration of computer systems today is a difficult task. Too easily, a computer user or administrator can make a simple mistake or lapse and misconfigure a system, causing instabilities, unexpected behavior, and general unreliability. Bugs in software that changes these configurations, such as installers, only worsen the situation. A self-managing configuration system should be continuously monitoring itself for invalid settings, preventing the bugs from harming the system. Unfortunately, while there are many constraints which can differentiate between valid and invalid settings, few of these constraints are explicitly written down, much less written down in a form usable by an automatic monitor. We propose an approach to automatically infer these correctness constraints based on samples of known good configurations. In this paper we present Glean, a system for analyzing the structure of configurations and automatically inferring four types of correctness constraints on that structure.

1. Introduction

Misconfigured systems are a significant source of problems in computer systems today. Whether they originate from user mistakes, application errors, or corrupted data, these misconfigurations cause serious problems for system reliability. In [9], Oppenheimer studies failures at three large Internet services and finds that configuration errors were the largest category of operator mistakes that caused end-user visible downtime. Studies of wide-area network systems indicate that misconfigurations in BGP are responsible for almost 3 of every 4 BGP routing announcements [8]; and that misconfigurations are a significant cause of extra load on DNS root servers [1].

We focus on configuration data stored in the Windows registry, including both application configuration information and operating system configuration. In [5], Ganapathi et al. study problems related to the Windows registry, and find that faulty configuration data in the registry can cause a variety of failures, such as causing general system instability, hiding criticial functionality from a user, or causing normal functionality to have unanticipated side-effects. These misconfigurations can occur for any number of reasons: an application installation or uninstallation might fail, leaving behind an inconsistent configuration or corrupting the configuration of other components of the system; a malicious or buggy program might corrupt a user's configuration; untested interactions between different versions of a library or program might cause configuration inconsistencies; or a user might not be aware of all of the side-effects of a setting and simply misconfigure the system.

An important part of a self-managing configuration system is prophylactic monitoring of the registry for possible problems. In their study of Windows registry problems [5], Ganapathi et al. find that over one third of the problems studied could be proactively detected and diagnosed by monitoring the registry for known problems and their signatures.

The difficulty in monitoring for problems is knowing what the monitor should be looking for. As shown in Figure 1, a monitor can use both positive and negative identification to detect possible problems. The positive identification approach is to monitor for known signs of problems. For example, if we know that a problem occurs when a particular configuration value is set to '0', we can watch for that change and possibly block it. Positive-identification depends on having a large list of known problems to watch out for. While this works well for detecting well-known problems, we also want to have the ability to detect potential problems that we have not seen before.

Negative-identification provides the ability to detect these unknown problems. Rather than looking for signs of known problems, we build constraints that describe what a good configuration looks like based on a training set of believed good snapshots of registries. When these correctness constraints are broken, we can assume that the registry is also broken, even though we may not have seen the specific situation at hand before. Unfortunately, very few of these correctness constraints are documented, and given that there are over 200,000 settings in the registry of a typ-

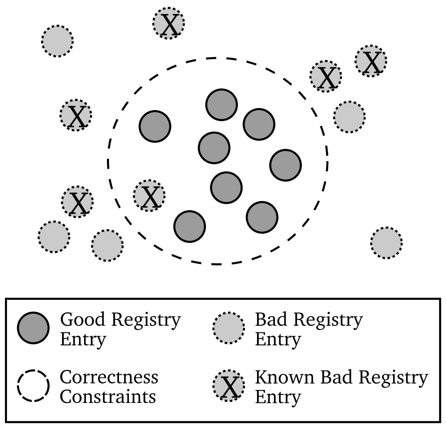


Figure 1. Positive detection looks for known indications of bad registries, but cannot detect problems not seen before. Negative detection applies correctness constraints that describe good registries and detects registries that do not meet these constraints as bad. Negative detection cannot, however, detect a bad registry that otherwise meets the correctness constraints.

ical machine, manually generating the constraints on these keys is simply not feasible.

In this paper we describe Glean, a system for automatically learning correctness constraints to use in negative-identification of problems. We present three contributions in this paper:

1. A method of discovering classes of configurations within a registry, imposing extra structure useful in inferring generalized correctness constraints.
2. Algorithms for inferring four kinds of correctness constraints on configuration settings.
3. An evaluation of these discovered constraints, based on an analysis of a Product Support Services database, describing problems encountered by customers and solved by the product support teams.

Section 1.1 presents a short background on the Windows registry. Section 2 provides an overview of Glean's approach to inferring consistency constraints. Sections 3 and 4 give the details for discovering configuration classes and learning correctness constraints. We present our results and evaluate our system in Section 5, and conclude with a discussion of related and future work.

1.1. Background: Windows Registry

The Windows registry provides centralized storage for information and settings about the hardware, operating system, applications, users, and user preferences on a Windows PC. The registry provides a hierarchical structure for settings, allowing keys to have subkeys and named values, similar to the directory and file structure of a file system, as shown in Figure 2. It is up to clients of the registry to decide how to organize their own settings, though some conventions are generally followed. For example, vendors usually place their user-specific application settings underneath the key \HKEY_CURRENT_USER\Software\[VendorName].

For the purposes of our analysis, we use a slightly modified representation of the Windows registry structure. Rather than having hierarchical sets of keys containing *<name,value>* pairs, we add the name of the value as a leaf key in our tree of registry keys and assign values directly to the leaves in our tree. This minor change simplifies our model, without causing loss of information contained in the registry.

2. Glean Approach

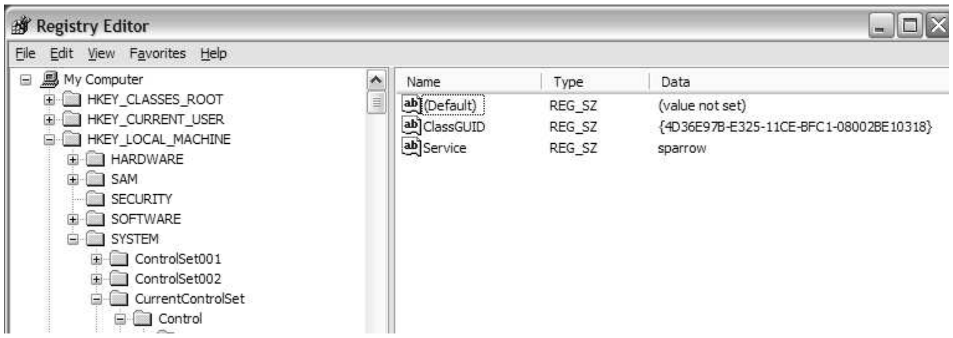
The Glean approach to generating correctness constraints is to analyze snapshots of "believed-good" registries in two stages. First, Glean searches for repeated groups of configuration settings in these registry snapshots to infer the existence of general classes of configuration settings. These *configuration classes*, directly identified by their common structure, form the basic unit of many of the constraints that Glean learns.

An example of a configuration class is the group of keys used to store information about each type of file (text files, GIF images, etc.) registered with the operating system. Each of these file type registrations uses many registry keys grouped together to describe, e.g., how to open the file. The same key structure is reused for each file type registration.

Once Glean finds these configuration classes, it searches for rules that describe the constraints and invariants on them. We hypothesize specific constraints based on the data in our registry snapshots, and validate our hypotheses against all our snapshots before fully believing it and asserting it as a constraint.

3. Configuration classes

Configuration classes are a natural first building block to discovering more about the structure of the information stored in the Windows registry. We know that many types of information stored in the registry, including software registrations, per-user account information, and hardware settings among others, are all repeated for each instance of the entity they describe. Discovering this extra structure within a registry allows us to create general configuration constraints that apply across all of the instances of a class.



Name	Type	Data
(Default)	REG_SZ	(value not set)
ClassGUID	REG_SZ	{4D36E97B-E325-11CE-BFC1-08002BE10318}
Service	REG_SZ	sparrow

Figure 2. Some of the hierarchical keys and values in a typical Windows registry. HARDWARE, SECURITY and SYSTEM are among the top-level keys, and each has several subkeys (though only SYSTEM is expanded to show its subkeys in this view). ClassGUID is the name of a value, and {4D36E97B...} is the actual content of that value.

We look for repeated groups of configuration settings that share a common hierarchical structure: two items in a configuration belong to the same class if more than a threshold amount of their substructure is identical.

Note that configuration classes ignore the ancestors of a key in the hierarchy. That is, whether or not two keys are in the same location in the hierarchy does not affect our decision to put the keys together in a configuration class. Basing our decision only on the substructure of keys allows us to define a finer-granularity of class and also allows us to detect configuration classes that are spread out across the registry in separate user accounts, backups of parts of the registry, etc.

The rest of this section describes how we discover configuration classes, name and describe the class, and how we name instances of these classes.

3.1. Class discovery algorithm

Glean uses data-clustering to infer additional structure about the keys and values stored in the registry. Data clustering algorithms are unsupervised learning algorithms that organize sets of objects by grouping together similar objects, as measured by some similarity or distance metric. In this context, data clustering can identify configuration classes by grouping together configuration settings with similar structures.

Our Glean prototype uses a hierarchical, bottom-up clustering method using arithmetic averages (UPGMA) and calculates the distances between clusters based on the simple convex average metric [7, 6]. We stop clustering when we meet a threshold distance. The complete algorithm for discovering configuration classes and extracting names for them is shown in Algorithm 1. The first step (before the data clustering is applied), is to filter out keys with little substructure. This filters out keys, such as leaf-nodes in the hierarchy of registry keys, with so little structure that they are unlikely to be part of a configuration class.

Once we have initialized each key into its own unary cluster, we begin running our data clustering algorithm. The $findClosestClusters()$ function searches for and returns the two clusters closest to each other. We merge that pair together, and add the combined cluster to our set of clusters. Once we have merged all clusters less than a threshold distance apart, we stop. At this point, the resulting clusters larger than $minclustersize$ are our discovered configuration classes.

Algorithm 1 Algorithm for data clustering and naming

```
load registry $R$
$clusters = \{k | k \in R, |k.subkeys| \geq minsubkeys\}$
{ignore keys that have too little substructure}
loop
  $pair = findClosestClusters(clusters)$
  if $pair.distance \leq thresholddistance$ then
    $clusters.remove(pair)$
    $newcluster = merge(pair)$
    $clusters.add(newcluster)$
  else
    exit loop
  end if
end loop
for all $c$ s.t. $c \in clusters$, $|c| \geq minclustersize$ do
{loop through resulting configuration classes}
  $name_c = \{s | \forall k \in c, s \in k.subkeys\}$
end for
```

The last step of the procedure is to extract the common substructure of the keys in each cluster as the name of the configuration class. To double-check that this class name is appropriate, we can verify that all or almost all of the keys in the registry snapshot that match this substructure are within

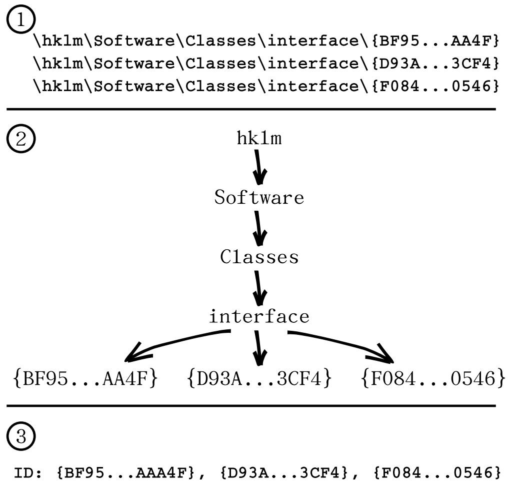

1
\hklm\Software\Classes\interface\{BF95...AA4F}
\hklm\Software\Classes\interface\{D93A...3CF4}
\hklm\Software\Classes\interface\{F084...0546}
2
hklm
Software
Classes
interface
{BF95...AA4F}
{D93A...3CF4}
{F084...0546}
3
ID: {BF95...AAA4F}, {D93A...3CF4}, {F084...0546}

Figure 3. By looking at the hierarchical difference in the names of the keys shown here, we can infer that $hklm$, $Software$, $Classes$, and $interface$ are not identifying names, and that the strings on the level of $BF95...AAA4F$ are.

the discovered cluster.

A more sophisticated naming system might only use those features which differentiate this class from others, in effect, reducing the size of the name without reducing its efficacy. Though this would be more efficient, our experiments have not yet warranted this complexity.

3.2. Naming instances

Once we have discovered the configuration classes, it is useful to also know how to name instances of registry keys within the class. This is especially important as we look for references to instances of this configuration class in Section 4.3. We can determine a superset of identifying strings by looking at the differences in the hierarchical key names. Figure 3 shows an example of this process. Here, we look at three keys that differ only in their last elements, presenting us with an obvious identifier for each key.

The problem of determining the identifying keys becomes more difficult when the configuration class is spread out across the registry in different locations in the hierarchy. For example, completely identifying user-specific config settings often requires using both the user's ID string, and a randomized ID together. One example is the configuration class formed by the keys:

```
\hku\S...797\....\shellnoroam\bags\55\shell
\hku\S...797\....\shell\bags\8\shell
\hku\S...451\....\shell\bags\3\shell
```

Using our heuristic of looking at the branchpoints in the tree represented by these keys, we see that the total set of identifying strings is $\{S...797, S...451, shellnoroam, shell, 55, 8, 3\}$.

In this example, the first branchpoint represents the user ID, and the final branchpoint is an identifying number. Together with the middle branchpoint, this creates a likely superset of the identifiers for a key. We call this a superset because it is possible that some of the branchpoints in the hierarchy may not actually be necessary to uniquely identify the keys. For instance, the difference between keys with $shell$ and $shellnoroam$ in the above example may not be significant. Unfortunately, without more information, we cannot determine which of these identifying strings are irrelevant, so we include them all.

4. Generating hypotheses

Once Glean has found the configuration classes in a registry, the next step is to search for the correctness constraints placed on the registry. Glean looks for both *internal* constraints that describe the valid structure of the values inside an instance of a configuration class, and the *external* relationships across configuration classes and arbitrary keys in the registry. In this section, we describe four kinds of constraints and how to generate them: size constraints, value constraints, reference constraints, and equality constraints.

Glean's general strategy for discovering these constraints is to first start with a template rule that describes the form of the constraint. Then Glean iterates over the sample registry snapshots and the discovered configuration classes and fills in the template to generate a hypothesis for a constraint. Glean validates this hypothesis against the data in each of the snapshots to ensure that it meets our confidence thresholds and either accepts or rejects the hypothesis as a valid constraint.

4.1. Size constraint

The size constraint specifies that, within a given configuration class, the value of a subkey always has a fixed size. This constraint is an example of an internal constraint, as it only refers to the structure within a single configuration class. Intuitively, the size constraint is likely to describe configuration settings that take a fixed form, even when the values of the form vary.

The template for a size constraint is "$\forall i \in C, |i.subkey| = x$", for a given size x and configuration class C. Using Algorithm 2 and a set of registry snapshots, Glean hypothesizes possible size constraints.

Algorithm 2 Inferring size constraints

```
for all $c$ s.t. $c \in set\ of\ configuration\ classes$ do
    for all $subkey$ s.t. $subkey \in name_c$ do
        if $\exists x$ s.t. $\forall k \in c, |k.subkey \rightarrow value| = x$ then {Found hypothesis}
            propose $|c.subkey \rightarrow value| = x$
        end if
    end for
end for
```

4.2. Value constaint

The value constraint is an internal constraint that declares that, within a configuration class, the value of a subkey always takes on one of a small set of values. For example, a value constraint easily describes situations where a key represents an option setting, such as a choice between $TRUE$ and $FALSE$.

The template for a value constraint is "$\forall i \in C, i.s \in \{x_1, x_2, ..., x_n\}$" for a small set of values X, a subkey s and configuration class C. To fill this template, Glean looks at all instances of a given type across our sample registries, and, for each subkey within the structure, sees what possible values it has. If the number of values is much less than the number of samples, Glean forms the appropriate hypothesis. In our implementation, we use $i < lg(n)$ as the threshold for determining whether $i \ll n$. Algorithm 3 summarizes this procedure.

Algorithm 3 Discovering values constraints

```
for all $c$ s.t. $c \in setofconfigurationclasses$ do
    for all $subkey$ s.t. $subkey \in name_c$ do
        if $|unique(i.subkey \rightarrow value|i \in c)| \ll |c|$ then {Found hypothesis}
            propose $c.subkey \in unique(i.subkey \rightarrow value)|i \in c$
        end if
    end for
end for
```

Currently, Glean only looks for constraints that limit registry values to one of a small number of enumerable values, and does not attempt to discover constraints that limit values to being within a continuous range of values, such as real values between $[0, 1.0)$.

4.3. Reference constraint

The reference constraint is an external constraint, and specifies that a particular key in the registry must always reference an instance of a particular configuration class. For example, a default printer setting should name the configuration settings for a printer registration.

The template for a reference constraint is "$k \in ID(i)|i \in C$", for some configuration class C, and where $ID(i)$ is the set of strings identifying the instance i of the configuration class C. Algorithm 4 shows how Glean infers a hypothesis from this template. Glean first creates a hashtable of all the values in the registry. As it adds the values to the hashtable, Glean does some preprocessing of the values, such as lower-casing all strings, to better match registry semantics. Glean also filters out any values that are too small, under the belief that values such as "1" are so common that they are more likely to generate false-positives than true constraints. Glean then iterates over the configuration classes in the registry, and makes a list of all the keys whose values match one of the instances of the configuration class.

While our internal constraints take advantage of the fact that most configuration classes are repeated many times—sometimes thousands of times—to provide a high confidence in a hypothesis, our external constraints only have one sample per registry snapshot. Because of this lower sampling, it is that much more important to cross-validate reference constraints across many registry snapshots. Our implementation of Glean hypothesizes reference constraints for registry snapshot being analyzed, then creates the final constraint only if the reference constraint exists in all of the snapshots.

Algorithm 4 Discovering reference constraints

```
Put all values in registry into a hashtable $t$ s.t. $t[v] = \{$ all keys with value $v\}$
for all $c$ s.t. $c \in setofconfigurationclasses$ do
    for all $id$ s.t. $\exists i \in c, id \in ID(i)$ do
        if $|t[id]| > 0$ then {Found hypotheses}
            $\forall k \in t[id]$, propose $\exists i \in c$ s.t. $k \rightarrow value \in ID(i)$
        end if
    end for
end for
```

4.4. Equality constraint

The equality constraint specifies that a set of keys in the registry must always have the same value, though it does not constrain what that value may be. The equality constraint is an external constraint, and is the only one of our correctness constraints that does not refer explicitly to configuration classes.

To discover equality constraints, Glean simply puts all the registry keys in a registry snapshot into a hashtable, hashed by their values. As in the previous subsection, Glean lower-cases strings as it places them in the hashtable, since many values, like hostnames and usernames in Windows, are case-insensitive. Again, Glean also filters out small values to avoid false matches. Glean then iterates over the hashtable, looking at all keys with a common value, and hypothesizes that any set of keys whose values are identical should always be equal to one another. Algorithm 5 summarizes this procedure.

To create a cross-validated equality constraint, Glean iterates over the hypotheses from one registry snapshot, and looks for mostly-identical hypotheses among the hypotheses from the other registry snapshots. Glean then intersects the keys on the left-hand-side of these equality constraint hypotheses to create a final equality constraint. If Glean cannot find similar hypotheses in each of the other snapshots, then it invalidates the hypothesis.

Algorithm 5 Discovering equality constraints

```
Put all values in registry into a hashtable $t$ s.t. $t[v] = \{$ all keys with value $v\}$
for all $v$ s.t. $t[v] \neq \emptyset$ do
    propose $\exists x$ s.t. $\forall k \in t[v], k \rightarrow value = x$
end for
```

5. Evaluation

We have built a prototype of Glean in C#, and use it to analyze eight registry snapshots from different Windows XP machines. All these machines were desktop deployments. Each of these registries had about 200,000 keys. Each of the performance numbers presented in this section is the mean of 3 successive runs of Glean on a 2GHz Intel Pentium IV machine with 500MB of memory.

5.1. Discovering configuration classes

The first step of our analysis of the Windows registry, discovering the configuration classes, began by filtering out registry keys with little substructure. This step left between 25,000 and 31,000 keys in each of our registry snapshots for us to analyze. We merged these keys into (on average) 1600 clusters. Only 1500 keys of the 25,000 were unique enough that they did not fall into any of our discovered clusters.

These 1600 clusters varied greatly in size, with the plurality of clusters having a small size (half contained only two keys), while the largest cluster had over 5500 keys grouped together. The mean size of the cluster was 15 keys; the median size was 6 keys. The largest class is identified by the signature:

```
(Default)
TYPELIB
PROXYSTUBCLSID32
PROXYSTUBCLSID
```

and maps to the keys in the location `\HKLM\Software\CLASSES\INTERFACE\*`. This location is where COM interfaces are registered, and these keys map interface identifies to the 16-bit and 32-bit versions of their interface libraries.

At this same location, we also discover a slightly different configuration class. The class defined below has 1700 instances. Even though it is stored at the same location, it differs significantly from the above. This configuration class represents 32-bit interfaces that do not provide a 16-bit interface library:

```
(Default)
PROXYSTUBCLSID32
NUMMETHODS
```

Other interesting configuration classes Glean discovers includes the file type registrations for video files (AVI, WMV, ASX, MPEG, ...), which is discovered as a class separate from the registrations for most image files (.GIF, .TIF, .JPEG, ...) and other formats. Glean also finds separate configuration class for each of <trust settings for various security certificates>, <security settings for different Internet zones>, <hotfix patch descriptions>, and many others. Continued spot-checking of the configuration classes Glean discovers shows us that the settings Glean discovers make intuitive sense.

For the rest of this paper, we arbitrarily chose the configuration classes discovered in one of our registries as the "canonical set" of classes to use when analyzing all our registries. Though we would have preferred to generate a canonical set by merging the configuration classes of many registries, time constraints kept us from implementing this feature.

Including the I/O time to read the registry snapshot and write the configuraton clusters to disk, generating these configuration clusters takes 4 minutes. The main resource constraint on our prototype is its unoptimized memory usage—it easily uses several hundred megabytes of memory to cluster the keys in a registry snapshot.

5.2. Generating constraints

After generating our configuration classes, we analyzed our registry snapshots to infer size, value, reference, and equality constraints. All together, Glean discovered 2785 size, 2706 value, 672 reference, and 1859 equality constraints.

Both kinds of internal constraints were generated and validated across three registry snapshots. Due to functional limitations of our initial prototype, the external constraints were generated from the analysis of a single registry, meaning they are less likely to generalize well across registries. Our prototype takes 3min 40sec to generate and validate the internal constraints, and 56sec to generate the external constraints.

Of the size constraints, 238 were rules declaring that the value of a key must be empty (size=0). Some of the more notable size constraints found included that the $CLSID$ subkeys (an abbreviation for the class id used to reference to COM objects) of most configuration classes had a size of 38, the correct length of a COM ID. Similar size constraints were found on keys that used different names, such as $EVENTCLASSAPPLICATIONID$, $APPID$, and $CLASSGUID$, to refer to class ids. Whereas a manually created constraint would likely only have looked for the well-known $CLSID$ and would have missed these others, Glean found all these automatically.

The value constraints show how Glean can infer clear and useful constraints on configuration settings. In one set of keys, located at \hklm\System\controlset*\services*, Glean discovers that the $TYPE$ subkey must have a value of 16 or 32, clearly refering to a distinction between 16-bit and 32-bit services. Glean also correctly generates value constraints on the perceived type and content type (or mime type) of the configuration classes for the various file type registrations described above. Glean correctly limits the video file types to being perceived as video files, and makes similar constraints on image files, audio files, compressed files, etc.

Among the reference constraints that Glean finds is one that declares that various "shell extensions" keys (that declare how files are opened in the Windows graphical interface) be limited to a class of COM registrations that provide details on context menu handlers, icon handlers, and other signatures of COM objects that are capable of handling file-related actions. Included among the equality constraints that Glean finds are all the various keys that store the hostname of a machine. Glean also discovers many registry keys that store user names.

We found that almost all of the constraints that we inspected to be reasonable. But, there are corner cases that cause Glean to behave poorly. For example, if a set of default user preferences is replicated within a registry, once for each user of the machine, it can quickly pass the required threshold of support to generate a rule that incorrectly constraints these preferences.

In particular, Glean is also vulnerable to poor sampling among its registry snapshots. For example, if Glean is fed registry snapshots from machines that do not have a particular application installed, Glean will obviously not be able to generate any constraints on that application's configuration settings. Worse, if Glean is fed bad registry snapshots, it can generate constraints that are too loose, and fail to detect problems. For now, the solution is to carefully choose the snapshots that Glean bases its constraints on, though the long-term approach is to scale up Glean's analysis techniques to analyze many more registry snapshots at once and/or use representative sampling of registries, and assume only that *most* of the snapshots are correct.

5.3. Detecting real errors

To evaluate Glean's ability to detect real configuration errors, we analyze a database of 43 serious registry problems gathered by the Strider project from Microsoft's Product Support Services knowledge base and e-mail case logs on customer issues and solutions. For each of these configuration errors, our database includes the offending registry key; whether the key's existence, absence, or an invalid value causes the error; and natural language descriptions of the symptoms and solution to the problem. Unfortunately, this database does not include enough information for us to determine the configuration class of the offending registry key. Instead, we rely on the configuration class of the key as found in our own registry snapshots. If the key does not appear in our snapshot, we pessimistically assume that Glean would not be able to determine the configuration class.

We evaluate each of these configuration errors against Glean's discovered consistency constraints. Overall, Glean successfully detects 33% (14/43) of these errors, with several being detected by multiple constraints. Our most successful constraint is the equality constraint, which detected 13 of these configuration problems. The size and enumeration constraints each detected 4 errors and the reference constraint detected 1.

The configuration errors that Glean's constraints did not catch fell largely into two categories. The first category of errors Glean missed were errors that added or removed keys to the registry but did not affect the value stored in a key. As Glean's constraints are mostly value-oriented, they did not notice these structural changes. This result indicates that a fruitful direction of future work would be to generate constraints on the sub-structure of keys. In a preliminary analysis, we find that if Glean had included a simple constraint that the configuration class of a key be stable over time, Glean would have detected 44% of these configuration errors.

The second category of errors Glean missed were those that changed the value of keys with unknown configuration classes. Part of this problem lies in our pessimism in assigning our configuration classes to the problem keys in our database. Glean would have detected several more errors if we had, for example, optimistically assigned configuration classes to new keys based on their location in the registry's hierarchy of keys. As discussed previously, a larger problem exists when Glean's training set comes from machines without the same applications as the machines Glean is meant to guard. For example, one of the errors Glean failed to detect was in a configuration for Microsoft Money v. 11, not installed in the snapshots used to train Glean.

6. Related work

The Strider project allows semi-automatic trouble-shooting of configuration problems [10], the second major part of a self-managing configuration system. Strider can trouble-shoot almost any user-perceived configuration problem, even ones where an otherwise valid configuration setting is at fault. In contrast, Glean's goal is to detect problems before the user notices them. However, Glean can only detect invalid settings.

Several projects have previously advocated building models of believed correctness based on observations of many systems and treating deviations from this model as likely problems. To find bugs in source code, Engler *et al.* instantiate rule templates into beliefs about correctness properties of the code; then marks code that contradicts these beliefs as a possible bug [4]. To improve the detection of application-level failures in complex Internet services, the Pinpoint project dynamically models normal patterns of communication and component behavior within the service and flags anomalous behaviors as possible faults [2].

In [3], Demsky and Rinard present a system which automatically detects and repairs data structure corruption, based on a specification of correctness properties. It may be fruitful to investigate using an expanded version of Glean to automatically discover the correctness properties required by Demsky and Rinard based on correct samples of the data structures in question.

7. Future work

There are several interesting research areas for future work related to Glean. An obvious step is to tightly integrate Glean with a dynamic monitoring system. Other areas of future work include developing constraints on configuration structures, adding specialized knowledge of common configuration primitives, such as IP addresses and filenames, and investigating statistical learning techniques, such as support vector machines, to more efficiently and robustly represent constraints.

Though this work has so far focused on detecting problems with the Windows registry, the basic principles behind Glean should be generally applicable to most configuration systems. Adapting the Glean system to generate correctness constraints for other systems, such as Unix /etc files, is an important avenue for future work.

8. Conclusions

Automatic inference of correctness constraints allows us to quickly extract a useful model of the constraints that govern the configuration of a system. Taking advantage of the hidden structure of configuration classes lets us generalize our constraints across groups of settings in a natural way and discover relationships otherwise hidden by the hierarchical representation of the registry.

Our experiments show that these automatic techniques can discover many of the consistency constraints that govern the validity of the Windows registry. Manual inspection of these constraints shows that they make intuitive sense. Applying these rules against a set of known registry problems shows that they can detect a significant percentage of problems based solely on snapshots of good registries.

References

[1] N. Brownlee, K. Claffy, and E. Nemeth. DNS Measurements at a Root Server. In *Sixth Global Internet Symposium*, San Antonio, TX, November 2001.

[2] M. Y. Chen, E. Kiciman, A. Accardi, A. Fox, and E. Brewer. Using runtime paths for macro analysis. In *9th Workshop on Hot Topics in Operating Systems*, Kauai, HI, 2002.

[3] B. Demsky and M. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, 2003.

[4] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, 2001.

[5] A. Ganapathi, Y.-M. Wang, N. Lao, and J.-R. Wen. Why PCs Are Fragile and What We Can Do About It: A Study of Windows Registry Problems. In *Proceedings of the International Conference on Dependable Systems and Networks '04)*, Florence, Italy, June 2004.

[6] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer-Verlag, 2001.

[7] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[8] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP Misconfi guration. In *Conference of the Special Interest Group on Data Communication (SIGCOMM)*, Pittsburg, PA, August 2002.

[9] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet services fail, and what can be done about it? In *4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, 2003.

[10] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, Y. Chun, and Z. Zhang. STRIDER: A Black-box, State-based Approach to Change and Confi guration Management and Support. In *Proceedings of Usenix LISA*, 2003.