NetPrints: Diagnosing Home Network Misconfigurations Using Shared Knowledge

Bhavish Aggarwal[‡], Ranjita Bhagwan[‡], Tathagata Das[‡],
Siddharth Eswaran[*], Venkata N. Padmanabhan[‡], and Geoffrey M. Voelker[†]

[‡]Microsoft Research India [*]IIT Delhi [†]UC San Diego

Abstract

Networks and networked applications depend on several pieces of configuration information to operate correctly. Such information resides in routers, firewalls, and end hosts, among other places. Incorrect information, or *misconfiguration*, could interfere with the running of networked applications. This problem is particularly acute in consumer settings such as home networks, where there is a huge diversity of network elements and applications coupled with the absence of network administrators.

To address this problem, we present *NetPrints*, a system that leverages shared knowledge in a population of users to diagnose and resolve misconfigurations. Basically, if a user has a working network configuration for an application or has determined how to rectify a problem, we would like this knowledge to be made available automatically to another user who is experiencing the same problem. NetPrints accomplishes this task by applying decision tree based learning on working and non-working configuration snapshots and by using network traffic based problem signatures to index into configuration changes made by users to fix problems. We describe the design and implementation of NetPrints, and demonstrate its effectiveness in diagnosing a variety of home networking problems reported by users.

1 Introduction

A typical network comprises several components, including routers, firewalls, NATs, DHCP, DNS, servers, and clients. Configuration information residing in each component controls its behaviour. For example, a firewall's configuration tells it which traffic to block and which to let through. Correctness of the configuration information is thus critical to the proper functioning of the network and of networked applications. *Misconfiguration* interferes with the running of these applications.

This problem is particularly acute in consumer settings such as home networks given the huge diversity in network elements and applications which are deployed without the benefit of vetting and standardization that is typical of enterprises. An application running in the home may experience a networking problem because of a misconfiguration on the local host or the home router, or even on the remote host/router that the application attempts to communicate with. Worse still, the problem could be caused by the *interaction* of various configuration settings on these network components. Table 1 illustrates this point by showing a set of typical problems faced by home users. Owing to the myriad problems that home users can face, they are often left helpless, not knowing which, if any, of a large set of configuration settings to manipulate.

Nevertheless, it is often the case that another user has a working network configuration for the same application or has found a fix for the same problem. Motivated by this observation, we present *NetPrints* (short for Network Problem Fingerprints), a system that helps users diagnose network misconfigurations by leveraging the knowledge accumulated by a population of users. This approach is akin to how users today scour through online discussion forums looking for a solution to their problem. However, a key distinction is that the accumulation, indexing, and retrieval of shared knowledge in NetPrints happens *automatically*, with little human involvement.

NetPrints comprises client and server components. The client component, which runs on end hosts such as home PCs, gathers *configuration information* pertaining to the local host and network configuration, and possibly also the remote host and network that the client application is attempting to communicate with. In addition, it captures a trace of the network traffic associated with an application run and extracts a *feature vector* that characterizes the corresponding network communication. The client uploads this information to the NetPrints server at various times, including when the user encounters a problem and initiates diagnosis. We enlist the user's help in a minimally intrusive manner to have the uploaded information labeled as "good" or "bad", depending on whether the corresponding application run was successful or not.

[*]The author was an intern at Microsoft Research India during the course of this work.

[†]The author was a visiting researcher at Microsoft Research India during the course of this work.

The NetPrints server performs decision tree based learning on the labeled configuration information submitted by clients to construct a *configuration tree*, which encodes its knowledge of the configuration settings that work and ones that do not. Furthermore, it uses the labeled network feature vectors to learn a set of *signatures* that help distinguish among different modes of failure of an application. These signatures are used to index into a set of *change trees*, which are constructed using configuration snapshots gathered before and after a configuration change was made to fix a problem. At the time of diagnosis, given the suspect configuration information from the client, the NetPrints server uses a *configuration mutation* algorithm to automatically suggest fixes back to the user.

We have prototyped the NetPrints system on Windows Vista and made a small-scale deployment on 4 broadband-connected PCs. We present a list of 21 configuration-related home networking problems and their resolutions from online discussion boards, user surveys, and our own experience. We believe that all of these problems and others similar to them can be diagnosed and fixed by NetPrints. We were able to obtain the necessary resources to reproduce 8 of these problems for 4 applications in our small deployment and also our laboratory testbed. Since we do not have configuration data or network traces from a large population of users, we perform learning on real data gathered for the applications run in our testbed, where we artificially vary the network configuration settings to mimic real-world diversity of configurations. Our evaluation demonstrates the effectiveness and robustness of NetPrints even in the face of mislabeled data.

Our focus in this paper is on the diagnostics aspects of NetPrints. We are doing separate work on the privacy, data integrity, and incentives aspects as well but do not discuss these here. Also, our focus here is on network configuration problems that interfere with specific applications but do not result in full disconnection and, in particular, do not prevent communication with the NetPrints server. Indeed, these subtle problems tend to be much more challenging to diagnose than basic connectivity problems such as full disconnection. In future work, we plan to investigate the use of out-of-band communication (e.g., via a physical medium) to enable NetPrints diagnosis even with full disconnection.

2 Related Work

We discuss prior work on problem diagnosis in computer systems and in networks, and how NetPrints relates to it.

2.1 Peer Comparison-based Diagnosis

There has been prior work on leveraging shared knowledge across end hosts, which provides inspiration for a similar approach in NetPrints. However, the prior work differs from NetPrints in significant ways.

Strider [19] uses a state-based black-box approach for diagnosing Windows registry problems by performing temporal and spatial comparisons with respect to known healthy states. It assumes the ability to explicitly trace what configuration information is accessed by an application run. Such state tracing would be difficult to do with network configuration, which governs policy (e.g., port-based filtering) that implicitly impacts an application's network communication rather than being explicitly accessed by applications.

PeerPressure [18] extends Strider by eliminating the need to identify a single healthy machine for comparison. Instead, it relies on registry settings from a large population of machines, under the assumption that most of these are correct. It then uses Bayesian estimation to produce a rank-ordered list of the individual registry key settings presumed to be the culprits. While this unsupervised approach has the advantage of not requiring the samples to be labeled, it also means that PeerPressure will necessarily find a "culprit", even when there is none. This outcome might not be appropriate in a networking setting, where a problem might be unrelated to client configuration. Also, PeerPressure is unable to identify *combinations* of configuration settings that are problematic.

Finally, Autobash [15] helps diagnose and recover from system configuration errors by recording the user actions to fix a problem on one computer and then replaying and testing these on another computer that is experiencing the same problem. Autobash assumes support for causality tracking between configuration settings and the output, which is akin to state tracing in Strider discussed above.

2.2 Problem Signature Construction

There has been work on developing compact signatures for systems problems for use in indexing a database of known problems and their solutions.

Yuan et al. [21] generate problem signatures by recording system call traces, representing these as n-grams, and then applying support vector machine (SVM) based classification. Cohen et al. [8, 9] consider the problem of automated performance diagnosis in server systems. They use Tree-Augmented Bayesian Networks (TANs) to identify combinations of low-level system metrics (e.g., CPU usage) that correlate well with high-level service metrics (e.g., average response time).

In contrast, NetPrints uses a set of network traffic features, which we have picked based on our networking domain knowledge, to construct problem signatures. Since these network traffic features tend to be OS-independent, NetPrints would be in a position to share

signatures across OSes. Furthermore, we use a decision tree based classifier to learn the signatures.

2.3 Network Problem Diagnosis

Active probing is widely used for diagnosing network problems. For example, Tulip [12] probes routers to localize anomalies such as packet reordering and loss. Such diagnosis relies on a model of how network elements such as routers operate. Likewise, several model- or rule-based engines have been developed for diagnosing configuration-related and other faults in wireless LANs. These include systems that rely on infrastructure-based monitoring (e.g., DAIR [5], Jigsaw [7]) and those that rely on cooperation among wireless clients (e.g., WiFiProfiler [6]).

Other diagnosis systems such as SCORE [11] and Sherlock [4] have modeled, and in some cases automatically discovered, dependencies between higher-layer, observable network events and the underlying network components. Formal methods have also been used to check the correctness of network configurations. For example, rcc [10] checks for a range of well-understood BGP properties.

In the context of NetPrints, it may be possible to construct such models for certain well-understood configuration settings (e.g., port-based filters), thereby allowing diagnosis based on active probing, rules, or formal methods. However, in general, configuration settings may not be documented or well-understood, hence NetPrints' black-box approach.

2.4 NetPrints Compared to Prior Work

We view NetPrints as being complementary to prior work on network diagnosis in two ways. First, NetPrints focuses on configuration problems that impact *specific* applications rather than on broad problems that impact the network infrastructure. Second, NetPrints uses a *blackbox* approach appropriate for arbitrary and poorly understood configuration information, avoiding the need for the network behaviour or dependencies to be modeled explicitly.

NetPrints draws inspiration from prior work on black-box techniques to diagnose systems problems and index them with signatures to enable recall. However, NetPrints' goal of identifying how to *mutate* a broken configuration to *fix a problem* leads us to use a different approach — decision tree based learning — compared to prior work. This is primarily because of the interpretable nature of a decision tree. Furthermore, NetPrints leverages domain-specific knowledge to construct *signatures* of networking problems. The diagnosis procedure in NetPrints is both state-based and signature-based.

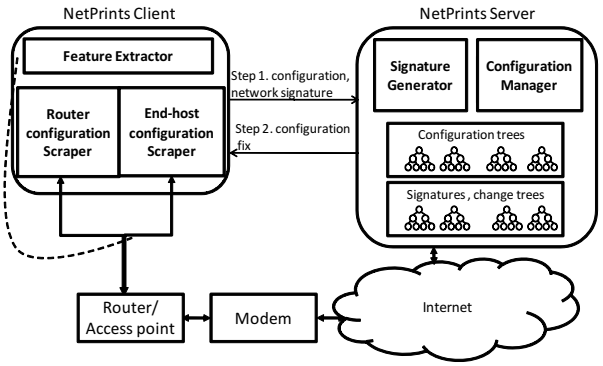

Figure 1: NetPrints system design

3 Overview of NetPrints Design

We begin with an overview of NetPrints, before turning to a more detailed discussion in the sections that follow.

Figure 1 depicts the client and server components of NetPrints, and their interaction. NetPrints has two modes of operation: "construction" and "diagnosis".

In the construction mode, the NetPrints server gathers configuration snapshots (Section 4) and network traffic features from NetPrints clients. This information is labeled as "good" or "bad" depending on whether the application run was successful or not. The NetPrints server, using this information, constructs a *configuration tree* (Section 5) that encodes its knowledge of which configuration settings work. It constructs a *change tree* (Section 7) based on the before and after snapshots of configuration changes that fixed a problem. Change trees are indexed by *network traffic signatures* (Section 6) that characterize how an application run fails. All these are constructed on a per-application basis.

When users experience a problem with an application, they invoke the diagnosis procedure. The NetPrints client, which runs on the user's machine, identifies which application to diagnose, either automatically (e.g., the application that last had focus) or with the help of the user. The client then gathers and uploads local configuration information and network traffic features, both labeled as "bad", to the NetPrints server (step 1 in Figure 1).

The NetPrints server performs diagnosis in two phases. In phase I, it uses the application-specific configuration tree to determine whether the client's configuration is problematic and, if so, identifies remedial *configuration mutations*, which it then conveys to the client (step 2 in Figure 1).

While configuration tree based diagnosis would work in many cases, it might fail, for instance, when there are "hidden" configuration parameters that impact a subset of the clients, so that the main configuration tree does not find anything amiss with the configuration of such clients (e.g., #4, #8, #10, and #12 in Table 1; see Sec-

#	App.	Router	Problem	Cause	Fix
1	VPN	WGR614	VPN Client does not connect	Stateful firewall was off	Turn on the stateful firewall
2	VPN	WRT54G	VPN drops connection after 3 minutes	(n/a)	Set MTU to 1350–1400, uncheck "block anonymous internet request", "filter multicast boxes" in router configuration
3	VPN	WRT54G	No VPN connectivity	No PPTP passthrough	turn on PPTP passhthrough
4	VPN	WRT54G	No VPN connectivity	double NAT, second NAT was dropping PPTP packets	Switch from PPTP server to SSTP server
5	File Sharing	any	Only unidirectional sharing	End-host firewall is not properly configured	Allow file sharing through all firewalls
6	File Sharing	WGR614v5	No file sharing	Client machine is on a domain, server machine is on workgroup	Put both machines either on the same domain or workgroup
7	FTP	any	Cannot connect to FTP server from outside home network	Port forwarding incorrect	Turn on port forwarding on port 21
8	FTP	WGR614	Cannot connect to FTP server at home	Client firewall blocking traffic, active FTP being used	Turn on firewall rule to allow active FTP connections
9	VPN server	WRT54G	PPTP server behind NAT does not work despite port forwarding and PPTP passthrough allowed	IP of server is 192.168.1.109, which is inside default DHCP range of router; router's port forward to IPs inside default range of router does not work	Use static IP outside DHCP range for server
10	Outlook	WRT54G	Outlook does not connect via VPN to office	Default IP range of router was same as that of the remote router	Change the IP range of home router
11	Outlook	WGR614	Router not able to email logs	SMTP server not configured properly	Setup SMTP server details in the router configuration
12	Outlook	Linksys	Not able to send mail through Linksys router; Belkin router works fine	MTU value too high for remote router, so remote router discards packets	Reduce MTU to 1458 or 1365
13	SSH	WGR614	SSH client times out after 10 minutes	NAT table entry times out	Change router or increase NAT table timeout
14	Office Communicator	WRTP54G	IM client does not connect to office	DNS requests not resolved	Turn off DNS proxy on router
15	STEAM games	WGR614	Listing game servers causes connection drops	Router misinterprets the sudden influx of data as an attack and drops connection	Upgrade to latest firmware
16	RealPlayer	BEFW11s4	Streaming kills router	Firmware upgrade caused problems	Downgrade to previous firmware
17	Xbox	WRT54G	Xbox does not connect and all games do not run	Some ports are blocked and NAT traversal is restricted	Set static IP address on Xbox and configure it as DMZ, enable port forwarding on UDP 88,TCP 3074 and UDP 3074, disable UPnP to open NAT
18	Xbox	WRT54G	Xbox works with wired network but not with wireless	WPA2 security is not supported	Change wireless security feature from WPA2 to WPA personal security
19	Xbox	WGR614	Not able to host Halo3 games	NAT settings too strict	Set Xbox as DMZ
20	IP Camera	DG834GT	Camera disconnects periodically at midnight, router needs reboot	DHCP problem	Configure static IP on the camera
21	ROKU	DIR-655	ROKU did not work with mixed b, g and n wireless modes	(n/a)	Change to mixed b and g mode

Table 1: Recent configuration-related problems in home networks.

tion 7 for an elaboration of #8). So in phase II, the NetPrints server uses a signature of the application problem to identify the appropriate change tree, which has been constructed by focusing specifically on such problematic cases. If the change tree is unable to diagnose the problem either, NetPrints gives up; it is possible that the problem is not configuration-related.

4 Configuration Scraper

The configuration scraper gathers configuration information from the local Internet Gateway Device (IGD) — which we loosely refer to as the local router — the local client host, and possibly also from a remote host and network.

4.1 Internet Gateway Configuration

The scraper gathers two categories of IGD information:
(i) *IGD identification information:* This information includes the make, model and firmware version of the device, which in most cases is a home router, although in some cases it could be a DSL or cable modem. The scraper obtains this information using the UPnP interface which is supported and enabled by default on most modern IGDs [16]. UPnP is a standard with which our client can obtain basic information such as the URL for the Web interface for the device, and the make and model of the device. However, if the router has UPnP turned off, we ask the user to manually input the IGD identification information. Note that the user will need to input this information only very rarely, i.e., when they install a new router that has UPnP turned off.
(ii) *Network-specific configuration information:* The IGD also includes configuration information such as port forwarding and triggering tables, MTU value, VPN pass-through parameters, DMZ settings, and wireless security settings. The scraper uses both the UPnP interface and the Web interface that most routers and modems provide to glean such configuration information. On some of the routers we tested, the port tables from the Web page and the port tables from the UPnP interface were not kept consistent with each other. Consequently, we scrape and combine the tables via both interfaces. Some router firmware versions also allow us to scrape the maximum NAT table size and the per-connection timeout for each table entry. These fields can be particularly useful in diagnosing problems such as #2 and #13 in Table 1.

While the UPnP interface gives us access to only device-identifying parameters and the UPnP port forwarding and port triggering tables, the Web interface is richer but not standardized across routers.

In particular, there is no standardized way for parsing the HTML to extract the (key,value) pairs defining the configuration. To address this problem, we make the observation that each configuration Web page of the device is typically an HTML form that includes a "submit" operation. We invoke this operation programmatically on each configuration Web page. Doing so causes the creation of an HTTP POST request containing all of the (key,value) pairs in an easy-to-parse form. For example, the body of the POST request might contain: `submit_button=index&dhcp_start=100&dhcp_num=50&dhcp_lease=1440`. It is then straightforward to extract the various DHCP-related configuration settings from this string.

While scraping Web forms, the NetPrints client asks for the user name and password set on the router. The user will need to input this information once, after which a cookie within the NetPrints client will remember the input to use every time it scrapes the Web interface of the router. Note that no such information is needed for the UPnP-based scraping.

4.2 Local Host Configuration

There is also much configuration information of relevance to network operation on the local client host itself, such as whether the network connection is wired or wireless, whether TCP window scaling is on or off, and end-host firewall rules. We currently scrape all interface-specific network parameters, TCP-specific parameters and firewall rules from the end-host. Our implementation uses the `netsh` utility available on Windows operating systems to get this information.

4.3 Remote Configuration

In general, the configuration of the remote host and network also impacts the health of network applications. In some cases, the configuration information at the remote end may be inaccessible to us (e.g., the remote host might be a server in a different administrative domain). In other cases, however, the remote host might be under the control of the same user as the local host. One example is communication between a client and a server on the same home network, say as part of a file or printer sharing application. Another example is when a user tries to access a service running in their home network from an external location, such as a user in their workplace accessing their home FTP server.

If the user installs the NetPrints client on the remote host as well, then, using simple password-based authentication, the local NetPrints client can obtain remote host and network configuration information. For every application, the NetPrints client keeps track of all remote hosts that it accesses or tries to access and, if the remote site runs NetPrints under the same administration as the local NetPrints client, the local client collects remote configuration information.

The impact of remote configuration on the health of a networked application can vary. In some instances, a

problem may arise because of misconfiguration at the remote end. For example, if the remote network blocks access to port 21, attempts to connect to an FTP server on that network would fail. In other instances, the remote configuration may not be problematic *per se*. Rather, it is the mismatch between the local configuration and the remote configuration that is problematic. For instance, while some users might be able to access a file server, others may not be able to because their credentials are not included in the access control list (ACL) on the server. In other words, there is a mismatch between the local configuration (the local user's credentials) and the remote configuration (the ACL on the server).

Once the remote configuration information has been obtained, it is incorporated into NetPrints' diagnostics procedure in the same manner as local configuration information. The one exception, which requires some additional pre-processing, is incorporating the mismatch between local and remote configurations, a problem we turn to next.

4.4 Composing Configurations

Since it is the combination of local and remote configurations that matters in some cases, we introduce new, composite configuration parameters that are derived by combining local and remote configurations parameters. Conceptually, a composite parameter, C, is a Boolean derived by applying a comparison operator, $\bigotimes$, to the local parameter, L and a remote parameter, R. That is, $C = L \bigotimes R$.

The specific comparison operators we focus on are equality "$=$" and set membership "$\in$". For example, if the local Windows workgroup $L1$ and the remote Windows workgroup $R1$ are the same, then $C1$ = 1. Else, $C1$ is set to 0. Another example is of checking whether the local username $L2$ is part of the remote ACL $R2$ for a file sharing application. If it is (i.e., $L2 \in R2$), the corresponding composite parameter $C2$ is set to 1.

4.5 Reducing Composite Parameters

Blindly comparing all pairs of local and remote configuration parameters results in an explosion in the number of composite parameters, most of which would be meaningless (e.g., a comparison of the local user name with the DHCP setting on the remote router). To limit the number of such composite parameters, without requiring an understanding of the semantics of the parameters, Netprints (1) only uploads composites that explicitly match, and (2) excludes parameters that exclusively have one value from the learning process.

In our experimental setup, the configuration scraper captures roughly 500 configuration parameters from the router and 2100 from the end-host, at each of the local and remote ends. This yields an additional 1500 composite parameters, after reduction is applied, and hence a total of (2100+500)x2+1500=6700 parameters.

5 Configuration Trees

Based on the labeled configuration information obtained from clients, we construct per-application decision trees, called *configuration trees*, which encode NetPrints' learning of which parameter settings work and which do not. We start with a brief introduction to decision trees and then turn to how NetPrints constructs configuration trees and uses these for diagnosis.

5.1 Decision Trees

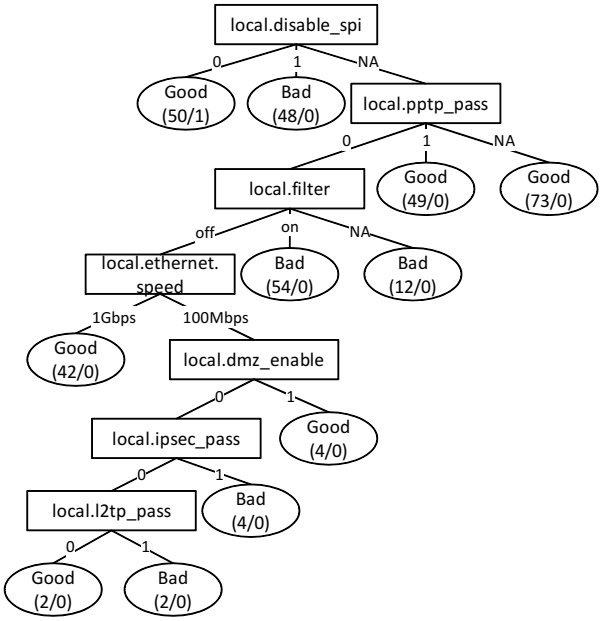

Figure 2: Configuration tree for the VPN client application discussed in Section 9.2.

NetPrints uses decision trees as a basis for performing configuration mutation. A decision tree (see Figure 2 for an example) is a predictive model that maps observations (e.g., a client's network configuration) to their target values or *labels* (e.g., "good" or "bad"). Each non-leaf node in the decision tree corresponds to an attribute of the observation, and the edges out of the node indicate the values that this attribute can take. Thus, each leaf node corresponds to an entire observation and carries a label. Given a new observation, we start at the root of the decision tree, walk down the tree, taking branches corresponding to the individual attributes of the observation, until we reach a leaf node. The label on the leaf node identifies configurations as "good" or "bad".

There are several algorithms for decision tree learning. We chose a widely-used algorithm, C4.5 [14], which builds trees using the concept of information gain.

The C4.5 tool starts with the root, and at each level of the tree chooses the attribute to split the data that reduces the entropy by the maximum amount. The result is that the branch points (i.e., non-leaf nodes with multiple children) at the higher levels of the tree correspond to attributes with greater predictive power, i.e., those with distinct values or ranges corresponding to distinct labels.

When the training data is noisy (e.g., it contains mislabeled samples) or there are too few samples, there is the risk that the above algorithm will over-fit the training data. To address this concern, C4.5 also include a pruning step, wherein some branches in the tree are discarded so long as this does not result in a significant error with respect to the training data (a process called generalization). C4.5 uses a confidence threshold to determine when to stop pruning. In our implementation, we use the default threshold. A consequence of pruning is that, if the number of samples is insufficient, these samples will not be reflected in the decision tree.

A decision tree has two key properties. First, it enables classification of observations that include both quantitative and categorical attributes. For example, the decision tree in Figure 7 includes quantitative attributes such as the WAN MTU and categorical attributes such as the security mode. Second, a decision tree is amenable to easy interpretation. It not only enables classification of observations, it also helps identify in what minimal way an observation could be *mutated* so as to change its label (e.g., from "bad" to "good"). We elaborate on this property in Section 5.4. The interpretability of decision trees, in particular, makes it an attractive alternative to SVMs or Bayesian classification.

5.2 Labeling Configuration Information

As explained in Section 4, the NetPrints client extracts configuration information from the local host and network as well as from the remote end. Before this information can be fed to the NetPrints server, it has to be labeled as either "good" or "bad", depending on whether the application in question was working or not. In general, it is hard to determine automatically whether an arbitrary application is working well. We sidestep this difficulty by enlisting the help of the human user to label the application runs. If we assume that the majority of users are honest, then most of the configuration information submitted to the NetPrints server will be labeled correctly. As we discuss in Section 9.6, decision tree based learning employed by the server is robust to mislabeling to a large extent. Also, in Section 10.1, we discuss ways of reducing the burden of labeling on users.

5.3 Configuration Manager

The configuration manager at the NetPrints server uses the labeled configuration information submitted by clients to learn and construct *per-application configuration trees*, using C4.5. The tree comprises decision nodes, which are branch points, and leaf nodes, which correspond to "good" or "bad" labels. A path from the root to a "good" ("bad") leaf node indicates the parameter settings for a working (non-working) configuration.

Figure 2 shows an example of such a configuration tree that we generated for the Microsoft Connection Manager VPN application [13] using configuration information from clients using several different router devices (see Table 5). We note that the `local.disable_spi` attribute (corresponding to whether stateful packet inspection (SPI) is disabled) is the clearest, even if not a perfect, indicator of whether a configuration is good or bad. So it is at the root of the configuration tree.

Note that a decision node in the configuration tree may have a branch labeled NA (not applicable), in addition to branches corresponding to the various parameter settings (e.g., 0 and 1 with `local.disable_spi`). The NA branch is needed since some parameters may be absent in particular routers.

Currently, the decision tree algorithm we use does not allow for incremental training of the trees, hence we use a cache of configurations to perform the training at each step. However, incremental update based algorithms exist [17] and we plan to evaluate these in future work.

5.4 Misconfiguration Diagnosis

When users experience application failure, they initiate the diagnosis procedure on the NetPrints client. The NetPrints client scrapes and submits its suspect configuration information to the NetPrints server for diagnosis. At the server end, the configuration manager starts at the root and walks down the configuration tree corresponding to the application that the user is complaining about. If it ends at a "bad" node, it means that the client's configuration is known to be non-working. On the other hand, if it ends at a "good" node, it means that the configuration tree is unable to help with the diagnosis, a case we consider in Section 7.

If the client's configuration corresponds to a known "bad" state, then the goal of diagnosis is to identify the *configuration mutations* that would move the configuration to a known "good" state. In general, there would be multiple "good" leaf nodes, so which one should we mutate towards?

Intuitively, we would like to pick the mutation path that is easiest to traverse. The easiest path is not necessarily the one with the fewest changes. The difficulty of making the changes also matters. For example, changing the router hardware (say switching from a Linksys router to a Netgear router) would likely be more difficult than modifying a software-settable parameter on

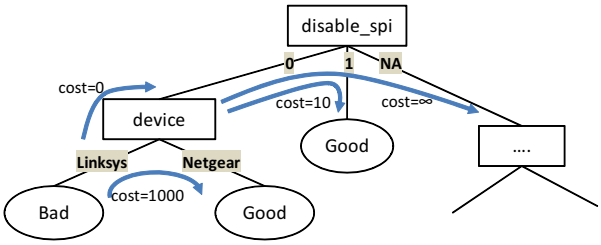

Figure 3: Illustration of the costs of different configuration mutations.

the router because of the costs involved. Even among software-settable parameters, some changes might be less desirable, and hence more difficult to make, than others. For example, putting the client host on the DMZ, and thereby exposing it to external traffic, would likely be less desirable than say enabling port forwarding for a specific port.

To determine the degree of difficulty automatically, NetPrints records the frequency with which various configuration parameters are modified across all clients. It might find, for instance, that the `disable_spi` parameter is modified 100 times as often as the `device` is. We quantify the cost of a mutation as the reciprocal of the change frequency, possibly scaled by a constant factor, of the corresponding configuration parameter. We might record some spurious changes, say when a mobile client moves from one network to another and mistakenly thinks that its router `device` and various configuration settings have "changed". However, we can counter the effect of mobility by hard-coding the fact that changing routers is a low-frequency, and therefore high-cost, change. Thereafter, when a client is mobile and associates with a new router, we infer that the corresponding changes in configuration detected by NetPrints are because the router changed, not because the user explicitly changed configurations. Hence we do not increase the change frequency of the parameters.

Figure 3 illustrates how the configuration tree is annotated with costs. The cost of changing the router `device` is 100 times greater than the cost of changing the `disable_spi` setting. Some mutations are impossible to effect, so the corresponding cost is set to ∞. For instance, it is not possible to set `disable_spi` to NA when the parameter does not exist on the router in question. Also, note that the cost is incurred only when a parameter is changed, hence the zero cost for merely walking up the tree.

Given the mutation costs indicated above, we compute the cost of moving from a "bad" leaf node to a "good" leaf node as the sum of the costs of the mutations on the path from the former to the latter. NetPrints recommends the set of mutations corresponding to the path with the lowest cost.

5.5 Going Beyond Configuration Trees

The per-application configuration trees help diagnose misconfigurations based on configuration information on which there is broad agreement across a large number of participating NetPrints clients. Basically, the configuration manager learns about the goodness or otherwise of various configuration settings based on *static snapshots* of labeled configuration information uploaded by clients.

However, as noted in Section 3, diagnosis based on the configuration tree would not work in the case of misconfigurations that are exceptions to the norm. Such exceptions could arise, for instance, from hidden configuration settings (as noted in Section 3) or from decision tree pruning (as explained in Section 5.1). In such cases, the configuration tree might suggest that the suspect configuration is "good" and hence not be in a position to suggest any mutations.

To address this issue, we introduce *change trees*, which seek to learn based on *dynamic* information, i.e., configuration changes. Furthermore, to reduce the chances of exceptions being buried by the mass, we use *network traffic signatures* to index the change trees.

Note, however, that multiple configuration errors could yield the same network signature, so a network signature is, in general, not as informative as the configuration information itself. Hence our approach is to use the configuration tree as the option, with the change trees indexed using network signatures as the fallback option.

We now discuss how NetPrints constructs network traffic signatures, and then turn to change trees.

6 Network Traffic Signature

We use a network traffic signature to characterize application runs. For instance, an application could fail because it is unable to establish a TCP connection (SYN handshake failure) or because the TCP connection is reset prematurely. The *network traffic signature* is used to distinguish between these failure modes. In essence, the signature records the *symptom* of the failure, which is used to index the change trees of the application, as explained in Section 7.

The basic approach is for the NetPrints clients to extract a set of *network traffic features* from a packet trace of the application run. The NetPrints server then applies learning on these features to identify the important ones, which are then included as part of the network traffic signature for that application.

6.1 Network Traffic Feature Extractor

The network traffic feature extractor characterizes the network usage of each application running on the client machine. In our current implementation, it uses the Winpcap library and the IPHelper API on Windows to tie all

#	Feature Description	Unit
1	TCP: Three SYN no response	5-tuple
2	TCP:RST after SYN, no data exchanged	5-tuple
3	TCP:RST after no activity for 2 mins	5-tuple
4	TCP:RST after some data exchanged	5-tuple
5	UDP: Data sent but not received	5-tuple
6	Other: Data sent but not received	src-dst IP addr pair
7	All: No data sent or received	all traffic

Table 2: Network traffic features and the unit of communication over which the feature is extracted. Each feature is maintained separately for inbound and outbound directions, except for "All", which is maintained for both directions together.

observed network traffic to the individual processes, and hence applications, running on the client machine. For each running application, it extracts a set of features by examining its network activity. These features form the *feature vector* for the application.

Table 2 lists the set of features we extract in the form of rules. Most of these features are maintained separately for the inbound (I) and outbound (O) directions, depending on whether the communication was initiated by the remote host or by the local host. While many of these features are extracted on a per-5-tuple basis (i.e., on per-connection basis for TCP), we combine the features across all connections of an application to compute the bits of the feature vector. Specifically, if *at least* one connection of an application satisfies any of these rules, the corresponding bit in the feature vector is set. Note that it is possible for multiple bits in an application's feature vector to be set. Also, while all of the features we consider at present are binary, the feature set could be expanded to include non-binary features.

We identified the set of features in Table 2 based on empirical observations of the ways in which an application's network communication may typically fail. The first four features in the table capture various kinds of TCP-level issues that we commonly see in malfunctioning applications. Several applications and services such as multimedia streaming, DNS and VPN clients use transport protocols other than TCP. For all of these, the lack of connectivity in one direction often indicates a networking problem. Consequently, we have included features #5 and #6 to capture the behavior of such applications. For both features, we use a timeout of 2 minutes: if no data is received for a period of 2 minutes, we interpret this as a possible problem and set the feature. Feature #7 characterizes a total loss of connectivity for an application using any transport protocol; problem #18 in Table 1, for instance, is a scenario in which our system would use this feature.

Finally, we briefly discuss two issues pertaining to the recording of network features for an application run. First, since the instance of an application could run for an extended period of time (e.g., a Web browser could run for days or weeks), we only consider network traffic features over a short window of time (typically a few minutes long) extending into the recent past. Second, extracting the network traffic feature for an application run requires capturing its traffic. One possibility is to run traffic capture continuously, which has the advantage that a record of the traffic will be available even when an application run failed.

To reduce the overhead of the NetPrints client with such traffic continuous capture, we split the network signature generator into two parts: a lightweight, continuously running component to capture selected packet headers and connection-to-process bindings, and a relatively more CPU-intensive component that creates the feature vector from the trace only when needed. Measurements of our implementation show that the overhead is low (0.8% CPU load) on a 1.8 GHz laptop PC running Windows Vista Enterprise, while streaming video over the Internet and simultaneously synchronizing email folders with the server.

6.2 Network Signature Generator

The NetPrints client records and uploads the feature vector for an application run to the NetPrints server, either when the user invokes NetPrints to complain about a non-working application or when the user is prompted, as explained in Section 5.2. In either case, the feature vector is labeled as "good" or "bad", just as the accompanying configuration information is. The NetPrints server then applies learning on the mass of labeled feature vectors for an application to identify the most significant features, i.e., ones that correspond most strongly to the fate of an application run. These significant features define the *network signature* of the application.

The signature generator, again, uses the C4.5 algorithm to learn the network signatures, which are represented as *per-application signature trees*. However, unlike with learning applied to configuration information, interpretability is not necessary for signature construction (since there are no mutations to perform), so we could have also used a different learning algorithm such as SVM. Figure 5 shows the signature tree generated for an FTP application, where 2 features, out of the 13 in all, are sufficient to capture the network problems seen.

7 Change Trees

As noted in Section 5.5, *change trees* are used as the fallback option when the configuration tree fails to diagnose a problem. To understand why configuration tree based diagnosis might fail, consider problem #8 in Table 1. The FTP server in question enables passive mode by default, so that all connections are initiated at the client end. However, in a small number of cases, the server may disable passive mode, i.e., only the server can initiate FTP data connections. The client will disallow these connections unless the client-side firewall has been configured to let them in. Note that the application-specific configuration parameter that captures the information that the server has disabled passive FTP is "hidden" from NetPrints since, in general, NetPrints is not in a position to scrape such parameters. Nevertheless, there are non-hidden configuration parameters (the firewall parameters on the client, in this instance) that could be manipulated to fix the problem.

Since the discriminating parameter is hidden, it is hard to tell apart the majority of clients that are configured for passive mode from the minority that are configured for active mode. So the majority prevails and the configuration tree learns to ignore the firewall settings since these are not of relevance for the majority of clients (i.e., FTP works for such clients regardless of the firewall settings). So when an active FTP connection to a client fails, the configuration tree would *not* find anything amiss with its configuration, i.e., it will find the configuration to be "good" and leave no scope for remedial action.

Change trees try to address this problem by isolating the cases where a traversal of the configuration tree ends up in leaf nodes labeled as "good" and then applying learning separately on these. For the purposes of this learning, the suspect configurations (which the configuration tree thinks of as "good") are labeled as "bad". Since we also need configurations labeled as "good" to perform learning, the NetPrints client in such cases looks for any out-of-band configuration changes that are made and, when such a change is detected, it prompts the user to determine whether the application problem has now been resolved. If and when the user indicates that the problem has been resolved, it uploads a "good" configuration to the NetPrints server.

The NetPrints server uses the C4.5 algorithm to learn a decision tree — the *change tree* — based on the change information: the "before" configurations labeled as "bad" and the "after" configurations labeled as "good". To isolate the relevant cases and minimize the mixing of unrelated problems, we use the network signature corresponding to application failure to index the change trees. So, in effect, each "bad" leaf node in the signature tree can point to a separate change tree.

Each change tree is also traversed the same way as the main configuration tree. If a traversal of the relevant change tree also ends in a leaf node labeled as "good", NetPrints gives up. It could be that NetPrints does not have sufficient information to identify the misconfiguration or that the problem is not configuration-related.

8 Summary of NetPrints Operation

In summary, NetPrints performs the following steps in the construction and diagnosis phases.

Construction Steps:

1) The NetPrints clients upload labeled configuration information and network feature vectors to the NetPrints server, either when users invoke NetPrints for diagnosis or are prompted by NetPrints (the latter happens for a small fraction of application runs).

2) The NetPrints server feeds the labeled configuration information into the C4.5 decision tree algorithm to construct an *application-specific configuration tree*. It feeds the labeled network feature vector to the same algorithm to learn an *application-specific signature tree*.

3) During the diagnosis phase (see below), if the traversal of the configuration tree with a suspect configuration terminates in a "good" leaf node, then this configuration, now labeled as "bad", is fed into the *application-specific change tree* construction procedure.

4) Furthermore, the NetPrints client prompts the user to determine if future configuration changes, if any, help restore the application to a working state. If so, the corresponding configuration, labeled as "good", is fed into change tree construction at the NetPrints server.

Diagnosis Steps:

1) When the user encounters a problem and invokes diagnosis, the NetPrints client uploads configuration information, along with the network feature vector for the affected application, to the NetPrints server.

2) The NetPrints server traverses the configuration tree with the suspect configuration submitted by the client. If this traversal ends in a "bad" leaf node, NetPrints identifies the set of configuration mutations, with the lowest cost, that would help move the configuration to a "good" state.

3) If the traversal of the configuration tree ends in a "good" leaf node, the NetPrints server first computes the signature of the failed application run based on the network feature vector submitted by the NetPrints client.

4) The NetPrints server uses the signature to identify the relevant change tree and then traverses this tree with the suspect configuration. If this traversal ends in a "bad" leaf node, then the NetPrints server uses the same procedure as indicated above to identify mutations.

5) However, if the traversal of the change tree ends in a "good" leaf node, the NetPrints server gives up.

Client		Server	
Config scraper	Feature extractor	Config manager	Signature generator
3159	701	1767	460

Table 3: Lines of code for NetPrints prototype.

9 Experimental Evaluation

Our experimental evaluation of NetPrints is based on the prototype we have implemented on Windows Vista SP1, using a combination C# and C++. Table 3 summarizes some information on the implementation; for C4.5, we used a standalone distribution [14].

We deployed the NetPrints client on 4 hosts behind separate broadband connections. Given this small scale of our current deployment, we used hosts on a separate testbed to scale up the effective size of the deployment, as we elaborate on below. The data gathered from the testbed was used in the "construction" phase of NetPrints during which the NetPrints server, which ran on a separate host, learnt the configuration, signature, and change trees. The "diagnosis" phase was initiated from one of the 4 broadband hosts and involved communication with the NetPrints server to perform diagnosis.

9.1 Setup and Methodology

We evaluated NetPrints with 4 applications: Microsoft's VPN client, a Perl-based FTP client, Windows Vista file sharing, and Xbox Live. These applications were run both on our testbed (construction phase) and a separate set of broadband hosts (diagnosis phase). Our testbed included a Windows Vista laptop (two in the case of the file sharing application), each running the NetPrints client, and also an Xbox 360 gaming console, all of which were uplinked via a home router and a DSL broadband modem. We also had 4 other hosts, including 2 at people's homes, on separate broadband connections, each running the NetPrints client from which diagnosis was initiated. Finally, for the FTP application, we also had an external machine running the client, not on a broadband network, that connected to one of the broadband hosts via the Internet.

For diversity, we used 7 different routers from Netgear, Linksys, D-Link, and Belkin (Table 5), in turn, as the home router in our testbed. To obtain greater diversity, as one might see with a large-scale deployment, we varied the configuration settings on these routers, rerunning the applications each time. Note that although we varied these configuration settings artificially, we ran the applications and NetPrints just as they would be run in the real world.

We identified 11 parameters (Table 4) and learnt variations in their settings based on a study of online discussion forums. Even with this subset of parameters, many configurations are possible (e.g., 4800 with the D-Link DIR-635 router). So for each application, we only experimented with a subset of these variations.

Router parameters:
MTU {1100, 1200, 1300, 1400, 1500 bytes}: supported by all routers except Belkin F5D7230.
VPN-specific parameters {on, off}: the D-Link router supports pass-through for IPSEC and PPTP, while the Linksys routers support these and also L2TP pass-through.
Stateful Packet Inspection (SPI) {on,off}: supported by all routers except Linksys WRT54G and Belkin F5D7230.
Wireless security parameters {none, WEP, WPA, WPA2}: all modes supported by all routers, except that the Netgear WGR614v5 does not support WPA2.
DMZ {on, off}: supported by all routers.
UPnP {on, off}: supported by all routers.
NAT type {symmetric, full cone, restricted cone}: only supported by Netgear WGR614v7 and D-Link DIR-635.
Port forwarding for FTP {on, off}: supported by all routers but only used for our FTP experiment.
End-host parameters:
Domain or Workgroup joined
Current user {Administrator, Guest, Everyone, other}
Windows Vista firewall rules {on, off}

Table 4: Parameters varied in our experiments

To automate the data collection process, we used AutoHotKey [1], a GUI scripting tool. To change configuration settings on the router, we used customized HTTP POST messages. To configure end-hosts, we manually changed the relevant parameters. For every configuration setting, we ran the applications and used simple application-specific heuristics to automatically determine whether the application worked (labeled as "good") or not ("bad"). These heuristics varied based on the application. For example, when the VPN client successfully connects, opening the VPN application's window displays the status of the connection. If the VPN connection was unsuccessful, then the same window shows the user an option to re-initiate the connection. Using AutoHotKey, we captured exactly which kind of message followed our attempt to set up the VPN connection, thereby determining if the application worked or not.

We recreated all of the problems related to VPN clients, file sharing, FTP, and the Xbox shown in Table 1, except for #2 and #6. In addition, our testbed itself presented new problems.

The diversity of configurations that we artificially induce in our testbed facilitates the construction of the application-specific configuration, signature, and change trees. However, it is hard to know how much diversity there would be in practice, in the absence of a large-scale deployment. Nevertheless, in Section 9.6, we demonstrate NetPrints' robustness to noisy data.

Finally, there is no standardized nomenclature for router configuration parameters. The parameter names vary across routers even when the functionality involved is the same. We avoid any manual steps to establish correspondence across routers or segregate information based on router model. If two router models happen to use the same parameter name, NetPrints will recognize and incorporate this in its learning process. Otherwise, it will treat the parameters as separate and unrelated. As standards such as HNAP [2] become prevalent, duplication would be reduced, resulting in more compact and better interpretable configuration trees.

9.2 Microsoft Connection Manager

The Microsoft Connection Manager (CM) [13] is a PPTP-based VPN client. For our evaluation, we used the 7 different routers in turn, varying the settings on each and then using CM to try connecting to an external VPN server. Table 5 shows the number of "good" and "bad" cases recorded with each router through this process.

Figure 2 shows the configuration tree for CM generated by the NetPrints server. Of all the configuration parameters, the algorithm picked `disable_spi`, `pptp_pass`, `filter`, `ethernet.speed`, `ipsec_pass` and `l2tp_pass` as the discerning ones. The numbers at every leaf node are of the form (x/y), where x is the total number of data points that the path from root to that leaf captures, and y is the number of misclassifications on that path.

We can explain the structure of the tree as follows. Only the Netgear routers support the specific `disable_spi` parameter. For these routers, CM works if `disable_spi` is *not* set and does not work if `disable_spi` is set, irrespective of the other parameter settings. On one of the runs involving the Netgear WGR614v5 router, CM failed even though `disable_spi` was not set, explaining the one misclassification on this path.

If `disable_spi` is not applicable, as for the Linksys, D-Link and Belkin routers, the next parameter that the tree learns is `pptp_pass`, which is available only on the Linksys routers. When `pptp_pass`=1, CM works with all three Linksys routers. If `pptp_pass`=0, there are further conditions, depending on the specific Linksys router. Finally, `pptp_pass`=NA for the D-Link and Belkin routers, through which CM works regardless of the settings. The `alg_pptp` parameter on the D-Link DIR-635, which is supposed to control PPTP pass-through, is apparently a no-op.

Next, the tree looks at `filter`, the stateful packet inspection parameter on the Linksys WRT310N and DD-WRT routers. The WRT54G does not support this option, so all configurations with `filter`=NA, i.e., all WRT54G configurations with `pptp_pass`=0, are bad.

The next parameter in the tree, on the `filter`=off branch, is `ethernet.speed`, an interface-specific parameter on the end-host. This is a little counter-intuitive but explainable. The only gigabit ethernet router we used was the WRT310N. Instead of using the model name to distinguish between the WRT310N and the DD-WRT routers, the C4.5 algorithm picked the ethernet speeds instead, since this has the same discriminating power as the model name in this case. This illustrates that learning is data-driven rather than based on intuition. If data were available from more routers supporting gigabit ethernet, we believe that C4.5 would have fallen back to the model name to differentiate among the various routers.

On the WRT310N (`ethernet.speed`=1Gbps), if `filter`=off, CM works irrespective of the other parameters. On the DD-WRT (`ethernet.speed`=100Mbps), CM's success depends on whether the client is placed on the DMZ. In particular, if the client is not on the DMZ, then CM works only if `ipsec_pass`=0 *and* `l2tp_pass`=0. We were unaware of this restriction until NetPrints constructed its configuration tree.

Next, we deployed the NetPrints client on 4 broadband networks using misconfigured Linksys WRT54G and DD-WRT, and Netgear WGR614v5 and WGR614v7 routers. When CM was invoked but the VPN connection failed, the user pressed the "diagnose" button on the NetPrints client. The NetPrints server then used its mutation algorithm to identify remedial configuration changes, which were then conveyed to the client. For the Netgear routers, the fix was to set `disable_spi`=0, whereas for the Linksys routers, it was to set `pptp_pass`=1. The NetPrints client automatically applies these fixes to the router using an HTTP POST to the corresponding Web form on the router.

This case study shows that NetPrints' configuration tree has *automatically* captured application behaviour with a large number of configuration settings across 7 routers and the client host, using a small number of branch points (only 7, in this case) in an intuitive representation. The tree also flagged configuration-related problems that we were unaware of previously.

9.3 Perl-based FTP Client

Users often set up FTP servers within their home networks so that they can have easy access to data on

Application	Netgear WGR614v5 ✓	Netgear WGR614v5 ×	Netgear WGR614v7 ✓	Netgear WGR614v7 ×	Linksys WRT54G ✓	Linksys WRT54G ×	Linksys DD-WRT ✓	Linksys DD-WRT ×	Linksys WRT310N ✓	Linksys WRT310N ×	DLink DIR-635 ✓	DLink DIR-635 ×	Belkin F5D7230 ✓	Belkin F5D7230 ×
Conn. Manager	25	25	24	24	13	12	34	20	50	40	48	0	25	0
FTP Client	–	–	156	254	309	169	–	–	–	–	67	89	46	26
Xbox	29	20	–	–	33	108	–	–	–	–	–	–	–	–

Table 5: A summary of the number of configuration settings we obtained from each router for VPN, FTP, and Xbox experiments. A "✓" lists the number of good configurations, and a "×" lists bad configurations. Cases where a particular router was not used with an application are marked with "–".

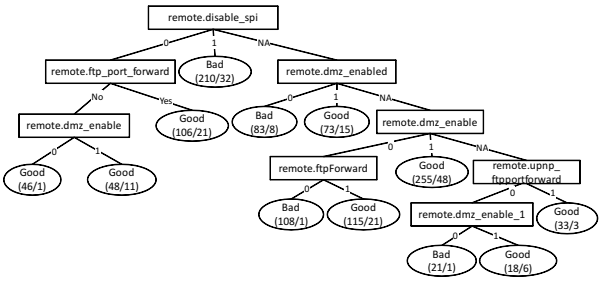

Figure 4: NetPrints configuration tree for the FTP client.

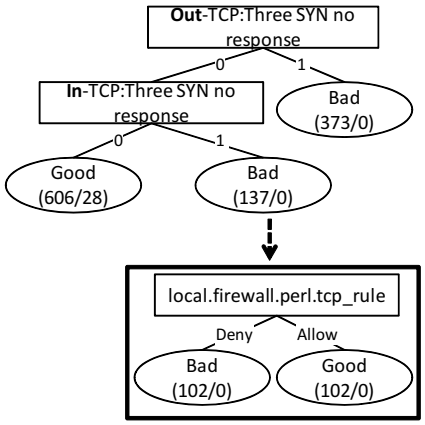

Figure 5: NetPrints change tree for the FTP client.

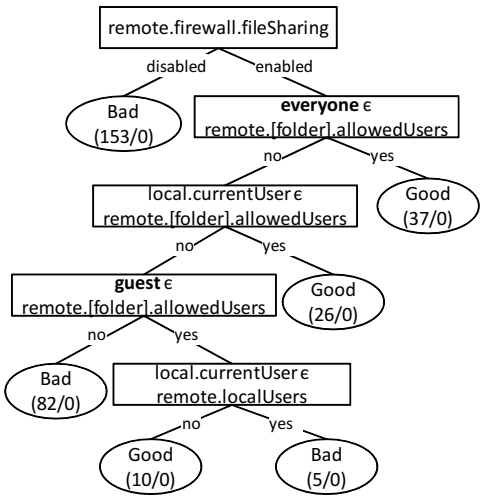

Figure 6: NetPrints configuration tree for file sharing.

their home computers from remote locations. However, the online discussions forums include several user complaints about the FTP service not running as expected when behind a NAT (e.g., #7 and #8 in Table 1).

To investigate #8, in particular, we evaluated NetPrints when a Perl-based FTP client running on a remote machine tries to connect to an IIS FTP server [3] running on a home network behind a NAT. Besides varying the router configuration settings, we also manually set and reset an application-specific parameter on the FTP client that determined whether the client used passive- or active-mode FTP. This corresponds to the hidden configuration example discussed in Section 7.

Figure 4 shows the NetPrints configuration tree, indicating the various server-side router settings (depending on the router model) needed for FTP to work. Since variable names for the same functionality vary based on the router, the tree has learnt three different variable names to capture the state of the DMZ (`dmz_enable`, `dmz_enabled`, and `dmz_enable_1`).

Note, however, that the misclassification count for most of the leaf nodes in the figure is significant. To understand why, consider the network signature and change trees shown in Figure 5. When the client uses active FTP, all of the server's connection attempts to the client fail, unless a firewall rule on the client host is enabled for allowing incoming TCP connections to the FTP client (this rule is disabled by default). The network signature for this problem has the "**Inbound**:Three SYN no response" feature set, since the client's firewall drops incoming connection attempts from the FTP server. Figure 5 also shows the change tree corresponding to this signature, which essentially says that the above firewall rule should be enabled.

While we used a Perl-based FTP client in this experiment for ease of automation, similar hidden configuration parameters exist in other clients. For example, IE 7.0 has a parameter to "turn off passive FTP connections", which, if set, would result in similar problems

and call for similar fixes as those discussed above.

9.4 Windows File Sharing

Home users often use file sharing within the home network. Online forums contain several complaints related to file sharing in Windows Vista, often caused by end-host configuration errors (e.g., #5 and #6 in Table 1).

To investigate these, we set up an experiment where a client host in our home network testbed tried to access a folder on a server host in the same home network. On both the client and the server, we varied the firewall settings, and the domain or workgroup that the machine was joined to. On the server, we varied the access control list (ACL) of users allowed to access the folder, and on the client, we varied the identity of the user who tried to access the folder. In all, we gathered data for 313 different configurations.

Figure 6 shows the configuration tree generated by NetPrints. In a nutshell, the configuration tree tells us that file sharing works if (a) the server-side firewall allows file sharing, and (b.1) either the special user "everyone" is a member of the folder's ACL or the current user on the client is a member of the folder's ACL, or (b.2) the special user "guest" is a member of the server's ACL list and the current user on the client is *not* a local user on the server.

This last point, b.2, is interesting since it suggests that the special user "guest" includes all users *except* the local users on the host machine. This is counter-intuitive since it means that guest users can, depending on the policy, have greater access than local users. We confirmed with experts within Microsoft that this is indeed expected behavior.

9.5 Xbox Live

Xbox Live [20] is a service that allows Xbox users to play multi-player games, chat, and interact over the Internet. One issue was that we could not run the NetPrints client directly on the Xbox since the consumer Xboxes are not user-programmable. For the sake of our experiments, we emulated a NetPrints client on the Xbox by instead running the client on a PC that is able to monitor all of the Xbox's network communication.

For this experiment, we gathered data for the Netgear WGR614v5 and the Linksys WRT54G routers, as indicated in Table 5.

Figure 7 shows the configuration tree generated by NetPrints. NetPrints learned three configuration rules. First, to make the NAT open, the router needs to enable UPnP. Second, Xbox 360 requires the router MTU to be greater than 1300 to enable connectivity to Xbox Live. Third, the Xbox wireless adapter could not connect to a wireless network if the security mode used was WPA2.

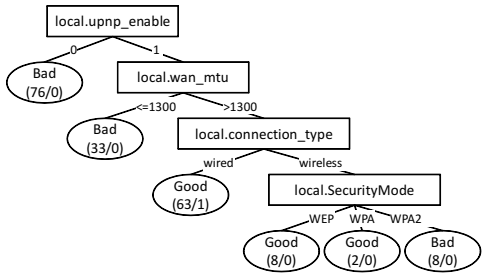

Figure 7: NetPrints configuration tree for Xbox Live.

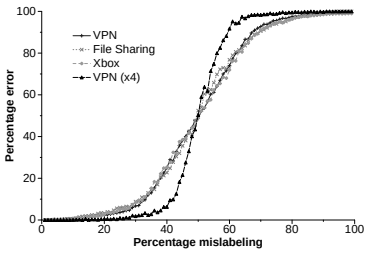

Figure 8: Sensitivity to mislabeled configuration data.

NetPrints' findings correspond to the suggested configuration fixes for #18 and #19 in Table 1, except for the MTU fix. We found out through support sites that Xbox Live requires the MTU to be set to 1365 bytes or larger. However, given that the data from our experiments, which formed the basis for NetPrints' learning, only had the MTU set to one of five values, the best inference we could make was that the MTU should be set to larger than 1300 bytes.

9.6 Robustness Tests

While our experiments have used clean and diverse data, in reality, configurations could be mislabeled and have limited diversity. Hence, we perform experiments to evaluate the robustness of the configuration trees to various conditions not found in our experimental data.

9.6.1 Mislabeled Configurations

In a deployed system, configurations uploaded to the server will not always be labeled correctly. Mislabeled configurations could potentially lead to troubleshooting a problem incorrectly, such as identifying a bad configuration as a good one. To evaluate the sensitivity of our configuration decision trees to mislabeling, we started with a known, correct set of labeled configurations and their associated decision trees. We then chose a random percentage p of those configurations and mislabeled them, flipping their labels from good to bad and vice versa. From this set with mislabeled configurations, we again generated decision trees and compared them with the original trees generated using correct labels.

Figure 8 shows the results of this experiment on the configurations for three applications: VPN (CM), File

Sharing and Xbox. The x-axis shows the percentage of mislabeling of configurations, and the y-axis shows the percentage of configurations incorrectly labeled in the decision tree based upon the mislabeled configurations. Each point represents the average across 100 trials. The VPN, File Sharing, and Xbox curves are similar and therefore difficult to distinguish. The VPN(x4) curve shows the effect of mislabeling for CM when the tree learning used four times as much data as from our testbed.

The results indicate that the applications are fairly resilient to mislabeling. While an insistence on no errors (0%) can only tolerate 2–4% mislabeling, allowing a 1% error (i.e., returning an incorrect configuration fix for up to 1 out of 100 diagnoses) allows tolerating 13–17% mislabeling. When more than 20% of configurations are mislabeled, though, the resulting decision trees overfit substantially, resulting in a high error rate. We also found that the effect of mislabeling diminishes significantly with a larger number of data points. For the VPN(x4) experiment, the tree tolerates 9% mislabeling (0% error) and 26% mislabeling (1% error), making it considerably more tolerant than the tree with the smaller configuration set.

Note that our methodology is not performing cross-validation on the data with training and testing sets. The reason is that we are not using the decision trees as classifiers. In other words, NetPrints does not use decision trees to classify or predict whether a configuration is good or bad — all configurations from the client already have labels ("good" or "bad") associated with them. The mislabeling experiment performs an extrinsic evaluation of the problem in terms of the utility of identifying an appropriate configuration mutation for a diagnosis in the face of incorrect labels.

9.6.2 Reduced Diversity

The configurations from our testbed experiments are roughly uniform in distribution in terms of the settings of the various parameters. In practice, the distribution is likely to be less diverse, with some settings much more prevalent than others (e.g., SPI might be disabled in 90% of configurations). In particular, the default configuration for a device, with an incorrect setting for a parameter, is likely to be prevalent, as is the resulting working configuration after correction.

Does low diversity further change the sensitivity of the decision trees to mislabeling? For each of the VPN, File Sharing and Xbox applications, we chose two configurations representing a default bad configuration and a default good configuration. We then introduced duplicates of those defaults to create low diversity. We varied the percentage of identical configurations from 0–95%, learnt the decision tree, and measured the extent of mislabeling similar to Section 9.6.1. For all of the applications, the effect of mislabeling was the same as with the original distribution of configurations.

10 Discussion

We now discuss a few broad challenges for NetPrints.

10.1 Reducing the Burden of Labeling

As noted in Section 5.2, NetPrints enlists the help of users to perform labeling of configurations (and also of network traffic traces). NetPrints employs several simple ideas to gather rich and accurate labeled data while minimizing the burden on users.

The labeling of "bad" configurations happens implicitly, as a by-product of a user invoking NetPrints for diagnosis when experiencing an application failure. Thus, it is only for having the "good" configurations labeled that the user's help must be enlisted explicitly.

However, prompting the user to label each run of an application as "good" or "bad" would likely be onerous and perhaps also provoke deliberately dishonest behaviour from an irritated user. So, in NetPrints, we only prompt each user for a small fraction of the application runs invoked by that user, with the expectation that, with a minimal burden placed on them, users would likely be honest while labeling. Given the participation of a large number of users, NetPrints is still able to accumulate a large volume of labeled configuration information, even while keeping the burden on any individual user low.

Furthermore, even the occassional prompting of a user is modulated so as to yield useful data with high likelihood. First, since the effective application of learning would require a mix of both "good" and "bad" data, users are prompted more frequently (with the hope of obtaining more data points labeled as "good") when the system is accumulating more "bad" data points because of users invoking NetPrints frequently to diagnose problems. Second, a user is more likely to be prompted when there has been a recent local configuration change. This policy increases the likelihood of novel information being fed into the learning process.

10.2 Preserving Privacy

Privacy is a key concern for NetPrints. Simply excluding privacy-sensitive configuration parameters such as usernames and passwords from the purview of NetPrints is not sufficient. Even the ability to tie back to the origin host (identified, say, by its IP address) configuration data uploaded to the NetPrints server could be problematic. For instance, knowledge of misconfigurations on a host could leave it vulnerable to attacks.

In ongoing work, we are working on a distributed aggregation system aimed at balancing two conflicting goals: enabling nodes to contribute data anonymously

while still enforcing tight bounds on the ability of malicious nodes to pollute the aggregated data. Thus, if a majority of nodes is honest, the aggregated data would be mostly accurate. While the details of this aggregation system are out of the scope of the present paper, we believe that NetPrints could directly use such a system.

10.3 Bootstrapping NetPrints

A participatory system such as NetPrints faces interesting challenges in bootstrapping its deployment. There is a chicken-and-egg problem in that users are unlikely to participate unless the system is perceived as being valuable in terms of its ability to diagnose problems, which in turn depends on the contribution of data by the participating users' machines. Even if this dilemma were resolved, there is still the challenge that users might resort to greedy behaviour, installing and running NetPrints only when they need to diagnose a problem and turning it off at other times, thereby starving the system of the data it needs to perform diagnoses effectively.

One could devise incentive mechanisms to encourage user participation. A complementary mechanism, which we are pursuing, is to bootstrap NetPrints using information learned via experiments in a laboratory testbed. This is similar to the methodology used for the evaluation presented in Section 9. While the richness of the testbed data would have a direct bearing on NetPrints' learning and hence its ability to diagnose problems, such an approach could help bootstrap NetPrints to the point where users perceive enough value to start participating.

11 Conclusion

We have described the design and implementation of NetPrints, a system to automatically troubleshoot home networking problems caused by misconfigurations. NetPrints uses decision tree-based learning on labeled configuration information and traffic features from a population of clients to build a shared repository of knowledge on a per-application basis. We report experimental results for a few applications in a laboratory testbed and a small-scale deployment. Our ongoing work focuses on scaling up the deployment and addressing privacy issues.

Acknowledgments

The authors would like to thank Monojit Choudhury for fruitful discussions on machine learning techniques. We would also like to thank our shepherd Kobus van der Merwe and the anonymous reviewers for their time and insightful comments regarding NetPrints and this paper.

References

[1] GUI scripting using the AutoHotKey tool. http://www.autohotkey.com.

[2] Home Network Administration Protocol. http://hnap.org.

[3] Microsoft Internet Information Services (IIS). http://www.iis.net.

[4] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards Highly Reliable Enterprise Network Services Via Inference of Multi-level Dependencies. In *SIGCOMM*, 2007.

[5] P. Bahl, J. Padhye, L. Ravindranath, M. Singh, A. Wolman, and B. Zill. DAIR: A Framework for Managing Enterprise Wireless Networks Using Desktop Infrastructure. In *HotNets*, 2005.

[6] R. Chandra, V. N. Padmanabhan, and M. Zhang. WiFiProfiler: Cooperative Diagnosis in Wireless LANs. In *MobiSys*, 2006.

[7] Y. Cheng, P. B. John Bellardo, A. C. Snoeren, G. M. Voelker, and S. Savage. Jigsaw: Solving the Puzzle of Enterprise 802.11 Analysis. In *SIGCOMM*, 2006.

[8] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. Chase. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *OSDI*, 2004.

[9] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, Indexing, Clustering, and Retrieving System History. In *SOSP*, 2005.

[10] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, 2005.

[11] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP Fault Localization via Risk Modeling. In *NSDI*, 2005.

[12] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet Path Diagnosis. In *SOSP*, 2003.

[13] Microsoft Connection Manager. http://support.microsoft.com/kb/221119.

[14] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kauffman, 1993.

[15] Y.-Y. Su, M. Attariyan, and J. Flinn. Autobash: Improving configuration management with operating system causality analysis. In *SOSP*, 2007.

[16] Universal Plug and Play Internet Gateway Device Specification. http://www.upnp.org/standardizeddcps/igd.asp.

[17] P. E. Utgoff. Incremental Induction of Decision Trees. *Machine Learning*, 4:161–186, 1989.

[18] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *OSDI*, 2004.

[19] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *LISA*, 2003.

[20] The Xbox Live Service. http://www.xbox.com/en-us/live/.

[21] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated Known Problem Diagnosis with Event Traces. In *EuroSys*, 2006.