Deja vu: Fingerprinting Network Problems

Bhavish Aggarwal[§], Ranjita Bhagwan[*], Lorenzo De Carli[†], Venkat Padmanabhan[*], Krishna Puttaswamy[‡]

[*]Microsoft Research India
[†]University of California, Santa Barbara
[‡]University of Wisconsin, Madison
[§]Olacabs.com

ABSTRACT

We ask the question: can network problems experienced by applications be identified based on symptoms contained in a network packet trace? An answer in the affirmative would open the doors to many opportunities, including non-intrusive monitoring of such problems on the network and matching a problem with past instances of the same problem.

To this end, we present Deja vu, a tool to condense the manifestation of a network problem into a compact signature, which could then be used to match multiple instances of the same problem. Deja vu uses as input a network-level packet trace of an application's communication and extracts from it a set of features. During the training phase, each application run is manually labeled as GOOD or BAD, depending on whether the run was successful or not. Deja vu then employs a novel learning technique to build a *signature tree* not only to distinguish between GOOD and BAD runs but to also sub-classify the BAD runs, revealing the different classes of failures. The novelty lies in performing the sub-classification without requiring any failure class-specific labels.

We evaluate Deja vu in the context of the multiple web browsers in a corporate environment and an email application in a university environment, with promising results. The signature generated by Deja vu based on the limited GOOD/BAD labels is as effective as one generated using full-blown classification with knowledge of the actual problem types.

1. INTRODUCTION

Network communication is an integral part of many applications. Therefore, network problems often impact application behavior. The impact on network communication depends on the nature of the problem. If the local name server is down, DNS requests will be sent but no responses will be received. On the other hand, if the firewall at the edge of a corporate network is blocking the `https` port, then SYN packets would be seen but not any SYNACKs.

We ask the question: can network problems experienced by applications be identified based on symptoms contained in the application's network packet trace? There are several advantages to looking for symptoms of network problems in a network packet trace. First, it is not intrusive unlike tracing on an end system itself (e.g., system call tracing). So we could monitor the health of applications running on several hosts without requiring access to the hosts themselves. Second, network communication represents the "narrow waist" of network applications. Many versions of an application (e.g., browser) and even OSes running on the end systems could exhibit consistent behavior at the level of network protocol messages, thereby leading to similar symptoms of problems at the network layer.

To answer the above question, we develop Deja vu, a tool to condense the manifestation of a network problem into a compact signature. Each signature encapsulates the symptoms corresponding to a particular problem. For instance, for a browser application that might encounter the problems noted above, there would be one signature corresponding to the local name server problem and a different one corresponding to the firewall problem. Although it might be tempting, based on these simple examples, to employ a rule-based approach to constructing signatures, such an approach suffers from the limitation of not being general enough to accommodate new applications or even existing applications whose behavior is not fully understood or documented.

Therefore, Deja vu uses a learning-based approach to constructing signatures. We extract a set of features from packet traces, using our domain knowledge to inform this. The features extracted correspond to protocols such as DNS, IP, TCP, HTTP, etc. For instance, there are features corresponding to the presence of a DNS request, DNS reply, HTTP error code, etc.

Once these features have been extracted, designing an algorithm to learn signatures is a key challenge. A standard classification approach, such as decision trees, would require labeled training data. Generating a training set with problem type-specific labels is onerous and could even be infeasible when the failure cause for a training run is unknown (e.g., a failure could occur in a remote network component). At the same time, an unsupervised learning approach, such as clustering, would be vulnerable to noisy data. For instance, features extracted from unrelated background traffic might still get picked for clustering.

To address this challenge, Deja vu employs a novel approach. For training, we only assume coarse-grained labels: GOOD when the training run of an application was successful and BAD otherwise. These labels can be determined based on the exhibited behavior of an application, without the need to know, in the case of BAD, the problem category. Then, by iteratively applying a decision-tree learning algorithm, Deja vu automatically learns different problem signatures for different categories of problems.

We evaluate the effectiveness of Deja vu in generating problem signatures for two classes of applications: multiple web browsers and an email client. For each application, we generate a training set by creating various error conditions. Similarly we generate a test set. We find that the problem signatures constructed by Deja vu based on the training set are able to classify the traces in the test set with 95% accuracy. In fact, the classification performed by Deja vu using just the GOOD and BAD labels is within 4.5% accuracy to that by a decision tree classifier operating with the benefit of problem category labels attached to traces. We also show how Deja vu learns new non-trivial problem signatures on-the-fly, which a rule-based approach would have missed. Finally we show the effectiveness of Deja vu's signatures in helping a human administrator match network packet traces to problems.

2. DESIGN OVERVIEW AND SCOPE

The input to Deja vu is a set of network packet traces, each coarsely labeled as GOOD or BAD. A GOOD trace corresponds to a working application run while a BAD trace corresponds to a non-working run. We believe that not assuming more fine-grained labeling is the right choice because we have found that applications often fail giving the same error messages for different networking problems, thereby not allowing a user to correctly differentiate between different bad runs. In our work, the GOOD/BAD labeling is performed by us in the lab, but we touch on alternative strategies in Section 7.

The coarsely-labeled traces are fed to Deja vu's feature extractor, which uses domain knowledge to extract a set of features, as discussed in Section 3. These *feature sets*, together with the GOOD/BAD labels, are then fed to Deja vu's *signature construction algorithm* discussed in Section 4. The novelty of this algorithm is that, although it is just given the coarse GOOD/BAD labels as input, it infers a sub-categorization of BAD corresponding to the different categories of problems that an application encounters.

Once Deja vu has learnt and associated signatures with problems, these could be used in a range of applications, helping to match the problems in a test trace to ones that have previously been seen and assigned signatures. We discuss two simple applications in Section 6.

Note that the extracted signatures can only be as good as the data input to the algorithm. The quality of the signatures therefore depends significantly on the choice of features, and the accuracy of the value of the features. Also, the scope of Deja vu is limited to problems that manifest themselves in network traces. There are several problems that applications experience which may not show as abnormalities in network traces. Deja vu does not address these problems. Consequently, the input features to our algorithm are extracted only from network traces, as we discuss in the next section.

3. FEATURES

In this section we describe what information we extract from the raw network traces and input to the Deja vu algorithm. As with any machine learning algorithm, Deja vu requires as input a set of *features*. The feature set extractor reduces a network packet trace to a compact set of features that summarizes the essential characteristics of the trace. This process also makes the input to Deja vu less noisy (e.g., features corresponding to unrelated background traffic are excluded) and strips it of privacy-sensitive information. For example, the actual packet payload is discarded except for some specific header fields in protocols such as HTTP and SMB.

The choice of features is key. Features that are too detailed often suffer from a lack of generality. To determine what kind of features to extract, we manually scrutinized and debugged traces for several networking problems. Using our domain knowledge and experience, we settled on the following broad categories of features to extract:

1. **Packet types:** Often, problems manifest themselves as the presence or absence of packets of a certain type. To capture this, we use *binary features* to record the presence or absence of certain packet types, where type is determined based on the packet header fields. By examining the headers of the packets contained in a trace, we set the corresponding binary features

in the feature set for the trace. For example, if a packet header indicates that it is an HTTP 200 Response (HTTP OK), we set the binary feature HTTP200_RESPONSE to 1. If no HTTP 200 Response is present in the trace, this feature is set to 0. We also set a special feature, NO_HTTP_RESPONSE, to 1 if no HTTP response of any kind exists in the trace.

2. **Sequence:** The presence or absence of a particular *sequence of packets* in a trace could also be symptomatic of a network problem. For example, a faulty trace might contain a TCP SYN packet but no corresponding TCP SYN/ACK packet, which might be indicative of a problem. To capture such effects, we define composite binary features, for example, one corresponding to the sequence TCP_SYN $\rightarrow$ TCP_SYNACK. We set this feature to 1 if and only if a TCP_SYN is followed by a TCP_SYNACK on the *same* connection. In all other cases, including when no TCP_SYN is seen, it is set to 0.

   We use our domain knowledge to trim the set of such composite features. For example, the sequence TCP RST $\rightarrow$ TCP SYN is not meaningful and therefore we do not record it.

3. **Aggregate features:** As we manually debugged site-specific problems by looking at traffic exchanged with a small set of websites, we found that the network traces for each site have very uniform characteristics in terms of aggregate quantities such as number of successful TCP connections, number of successful HTTP connections, number of bytes transferred, etc. While these features may be unimportant for generic networking problems, they are are particularly useful in the context of site-specific problem signatures for certain (popular) remote sites or services. For example, the `www.cnn.com` webpage has specific characteristics in terms of its layout and the number of objects on the page. The aggregate features can thus help capture the problem symptoms specific to this site. Also, the popularity of this site might make it worthwhile to learn site-specific signatures for it.

In our implementation, we have written feature extractors for 12 protocols using Microsoft Network Monitor's Parser API [1]. Table 1 shows the set of protocols and the number of features we capture from each protocol. For each application, we choose a subset of these protocols to determine what features to extract. Feature extractors for some protocols take as input a port number so that they extract features specific to traffic to and from that port. For example, when analyzing traces from browsers, we extract TCP-level features specific to port 80. We extract more features for higher-level protocols such as HTTP and SMB because these protocols provide richer error information in their headers than lower level protocols like TCP and IP. The feature extraction algorithm traverses each trace once to extract all the features and is therefore is not compute-intensive.

Protocol	No. of Features
ARP	3
DNS	4
HTTP	42
IKE/AUTHIP	4
IP	1
LLMNR	4
NBNS	4
RWS	4
SMB/SMB2	45
TCP	48
TLS	10
UDP	52

Table 1: Number of features for each protocol.

4. DEJA VU ALGORITHM

The Deja vu algorithm is used to identify signatures corresponding to the various problems experienced by an application. The algorithm takes as input the sets of features from multiple application runs, each of which is labeled as GOOD (if the run was successful) or BAD (if the run was unsuccessful). The goal, and the key challenge, is to work with these coarse labels to decompose the set of BAD runs into *meaningful* categories corresponding to the different problems experienced by the application.

This challenge sets the problem apart from standard problems of classification and clustering in machine learning. Unlike with classification, the categories of faults are *not* known in advance and hence we do not have (fine-grained) labels corresponding to the categories. Unlike with clustering, the coarse-grained GOOD/BAD labels do matter since our goal, specifically, is to sub-categorize the BAD runs. In contrast, clustering on the BAD runs to find categories might bunch together runs based on some extraneous features (e.g., the presence or absence of background noise). These extraneous features may be similarly present or absent in GOOD runs too, which means they are, in essence, inconsequential and should have no bearing on categorizing different kinds of BAD runs.

Consider the following example. Say half the traces for an application (both GOOD and BAD) have ARP requests, and the other half do not since the end-host caches ARP responses. Therefore, the presence or absence of a ARP request is not a symptom of a problem. However a conventional clustering algorithm on

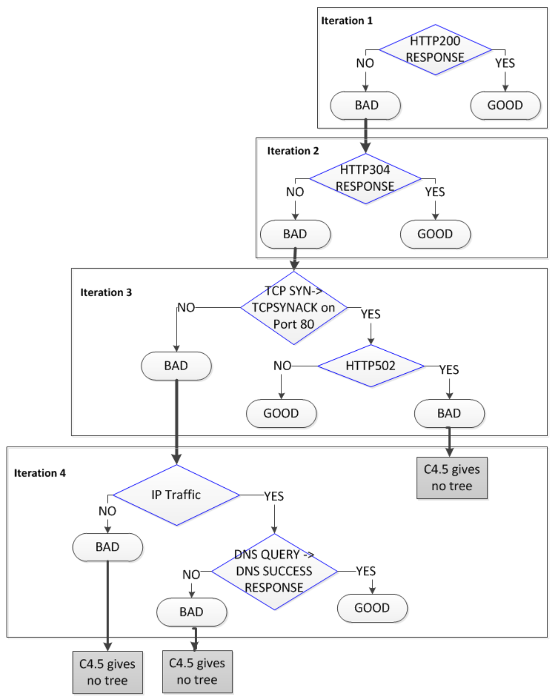

Figure 1: A signature tree learnt by Deja vu

the BAD traces will divide these traces into two categories based on the presence or absence of this feature. On the other hand, including the GOOD traces in our learning procedure will help us learn that the presence of the ARP request is inconsequential since half of the GOOD as well as the BAD traces have this feature, and the other half do not.

To address the challenge of learning the categories of problems from coarse-grained labels, we have developed a novel algorithm for Deja vu. We start by illustrating the algorithm with an example in Section 4.1, and then specify it in Section 4.2.

4.1 Example

We now illustrate the operation of the Deja vu algorithm by demonstrating how an example *signature tree* is constructed for a browser application. The main intent is to extract multiple problem signatures from the input feature sets. Toward this goal, the algorithm makes use of the C4.5 decision tree classifier([12]) in multiple iterations. The signature tree captures all the problem signatures for the application. Figure 1 shows an example signature tree.

In the first iteration, the Deja vu algorithm inputs the feature sets corresponding to all runs, labeled GOOD or BAD, to the C4.5 decision tree classifier. The classifier outputs the tree shown within the box titled “Iteration 1” in Figure 1. This tree tells us that:

If a trace does not have an HTTP 200 Response, it is BAD.

While this statement tells us something about a generic symptom that all the bad traces have, it does not help categorize the bad traces into *different sub-categories* or problems. So, we force the decision tree classifier to learn an alternative way of classifying the bad traces by removing the HTTP200_RESPONSE feature from all good and bad traces, and inputting the pruned feature sets to C4.5 in Iteration 2. The output of this iteration is shown in the box marked “Iteration 2” in Figure 1. This tells us that:

If the trace does not have an HTTP 304 Response, it is BAD.

However, in this iteration too, the decision tree classifies all bad traces into one category because it found yet another (novel) feature that is common to all bad traces. Hence, for iteration 3, we remove the HTTP304_RESPONSE feature from all good and bad feature sets and learn a decision tree again. In iteration 3, the tree has 2 leaf nodes labeled BAD:

If the trace (a) has no TCP SYN/SYNACK exchange on port 80, **OR** *(b) has a TCP SYN/SYNACK exchange on port 80 AND an HTTP 502 Response (proxy error), it is BAD.*

Hence, by iteratively removing features that exist in all BAD traces, the algorithm has succeeded in learning two different problem signatures and sub-categorizes the bad traces across them. The absence of a TCP connection characterizes the first signature (marked (a)), while an HTTP 502 response (proxy error) characterizes the second (marked (b)).

In iteration 4, we *divide* the BAD feature sets into two categories: one each for the two leaves labeled BAD in Iteration 3 above. The first category has no TCP SYN/SYNACK exchange on port 80, whereas the second category does but also includes an HTTP 502 response. With each category of bad feature sets, we separately repeat the procedure described in the previous iterations. Therefore, from the BAD feature sets in the first category as well as from all GOOD feature sets, we exclude the “TCP SYN $\rightarrow$ TCP SYNACK on port 80” feature and then rerun C4.5. On the other hand, from the BAD feature sets in the second category as well as from all GOOD feature sets, we exclude both the “TCP SYN $\rightarrow$ TCP SYNACK on port 80” feature and the “HTTP 502 Response” feature, and then rerun C4.5.

Iteration 4 for the first BAD category obtained in Iteration 3 gives us the tree shown in the box marked “Iteration 4” in Figure 1. This tree tells us that

If the trace (a) has no IP traffic, **OR** *(b) has IP traffic but no Successful DNS Query, it is BAD.*

The algorithm therefore splits the first BAD category from Iteration 3 into 2 sub-categories: the first has no IP traffic, and the second has IP traffic, but no successful DNS exchange. With these two sub-categories, we repeat the procedure we described in Iteration 3 of splitting the BAD traces into two sets, removing the discerning features from them, removing the same features from all GOOD traces, and inputting this new data to C4.5.

Both these runs of C4.5 yield trivial, one-node trees with no branches, which is the stopping condition for the Deja vu algorithm. Also, the second run of C4.5 in iteration 4 (corresponding to the second BAD category obtained in Iteration 3) yields a trivial, one-node tree. Hence, the signature tree is complete.

Having completed the process of growing the signature tree, we now prune the signatures to make them more concise. Note that this step is not necessary for correctness. However, in general, several features in signature tree constructed above could be redundant. For instance, in the example in Figure 1, the inclusion of **NO** HTTP304_RESPONSE immediately following **NO** HTTP200_RESPONSE is redundant since it does *not* help sub-classify BAD. Hence, we remove redundant features such as HTTP304_RESPONSE from all signatures.

Our final list of problem signatures corresponding to Figure 1 is:

1. **NO** HTTP200_RESPONSE **AND** TCPSYN80 $\rightarrow$ TCPSYNACK80 **AND** HTTP502_RESPONSE
2. **NO** HTTP200_RESPONSE **AND** **NO** (TCPSYN80 $\rightarrow$ TCPSYNACK80) **AND** **NO** IP_TRAFFIC
3. **NO** HTTP200_RESPONSE **AND** **NO** (TCPSYN80 $\rightarrow$ TCPSYNACK80) **AND** IP_TRAFFIC **AND** **NO** (DNS_QUERY $\rightarrow$ DNS_SUCCESS_RESPONSE)

These problem signatures are such that every bad trace matches only one signature. This characteristic is important in that it helps disambiguate between the seemingly large number of error conditions that occur in real-world applications.

If, instead of crafting these signatures through the Deja vu algorithm, we use a simpler, rule-based problem signature matching scheme which included the following rules among others for example:

1. HTTP502_RESPONSE
2. HTTP401_RESPONSE
3. **NO** IP_BACKGROUND_TRAFFIC

we might not get the level of discernment that Deja vu gives us. For instance, a web browser trace could contain an HTTP 401 error (Unauthorized), at which point the browser asks the user for a password. Once the user enters the password, the connection succeeds. This case demonstrates that just the presence of an HTTP 401 error does not necessarily indicate a problem. Our evaluation in Section 5.2.1 describes instances where Deja vu captured accurate problem signatures that a basic rule-based engine would not have captured. In fact, even a classifier-based approach with more fine-grained problem labels was unable to learn some problem signatures that Deja vu learned.

4.2 Summary of Algorithm Steps

We summarize the main steps of the Deja vu algorithm in the following 4 steps.

- **Step 1:** Extract all feature sets from the network traces, and assigns a label – either GOOD or BAD – to each feature set.
- **Step 2:** Input the labeled feature sets to the C4.5 algorithm, which yields us a decision tree.
- **Step 3:** If C4.5 does not give us a tree, stop the algorithm. Else, find all BAD leaf nodes in the tree.
- **Step 4:** For each BAD leaf, find all the features and their values in the path from root to the leaf. For all BAD feature sets that have these features and the same values, and for all GOOD feature sets, we *remove* these features. With these reduced feature sets, we start from Step 2 again.

5. EVALUATION

Our evaluation focuses on the effectiveness of the signatures learned by Deja vu along two dimensions: (a) how Deja vu's signatures, learned just using the coarse-grained GOOD/BAD labels, compare with those learned by a classifier that has the benefit of fine-grained, problem-specific labels, and (b) how effective Deja vu is in categorizing data in a test set and learning new signatures.

5.1 Data Collection

Since obtaining network traces from the field, together with ground truth information on the presence and nature of failures, is challenging, we have evaluated Deja vu by recreating real network problems in two different live environments – a corporate network and a university network. This fault injection based strategy is similar to evaluation of algorithms in previous network diagnostics research [2, 6]. The failures that we recreated (described in the following) were selected by browser popular user forums and discussing with network administrators.

We have evaluated Deja vu with web browsers running in the corporate network, and with email clients running in a university network. We chose web browsers and email clients as applications to evaluate since these represent significant applications in the today's enterprises. For instance, web browsers are used not just to access the public web but also to access myriad intranet services (e.g. HR, payroll).

For each application, we recreated a mix of problem scenarios ranging from obvious and well-known problems to more subtle issues, and collected network traces of these scenarios. We manually injected the failures by either misconfiguring the applications, operating systems, or network components. For each application run, we recorded a network packet trace and labeled it as GOOD or BAD, depending on whether the application run was successful or not. Note that for a number of these problems, the root cause is not obvious to the user just from the message that the application provides, justifying the coarse-grained labeling of GOOD and BAD. In addition, to enable comparison with a classifier, we recorded fine-grained labels indicating the root cause of each failure. Note that these labels were *not* made available to Deja vu.

Next, we describe the specifics of the data collection procedure we used for the browser and email datasets.

5.1.1 Browser

We used five different browsers – Google Chrome 5.0.3, Safari 5.0.1, Firefox 3.6, Opera 10.53, and IE 8 – to collect traces for various browser-related problems from within a corporate network. To collect the traces, we ran these browsers on three machines, each with a different OS — Windows 7, Ubuntu Linux 9.10, and Mac OSX 10.5.8 — subject to the availability of each browser on these OSes.

We reproduced each of the problems listed in Table 2 in our setting on each available browser + OS combination. For each such combination, we collected a set of GOOD traces and BAD traces for each of the problem scenarios listed in Table 2. In some cases, the problem was applicable to only a specific browser, so we collected BAD traces only for that particular browser. In all, we collected 878 traces for this dataset: 307 GOOD traces and 571 BAD traces. Note that, for lack of space, in the following we discuss signatures only for 7 of 11 the cases listed in table 2.

Features: We use our domain knowledge to determine which protocols are relevant to browsers, and extract features pertinent to this set of protocols. The feature extractor summarizes each browser trace using features specific to HTTP, TCP on port 80, all name resolution protocols in use (DNS, Netbios, LLMNR), and generic features that can help capture low-level problems, such as the presence or absence of background IP traffic.

5.1.2 Email

We collected email related problem traces in a university network using the Thunderbird 3.1.2 client running on three machines, each with a different client OS: Mac OSX, Windows XP, and Ubuntu 9.10. The Windows XP and Ubuntu traces are from the same university network, while Mac traces are from a machine connected to the university's residential network via a wireless access point. The clients connected to one of two email servers, each of which supported IMAP and SMTP over SSL.

We used these configurations to collect network traces for problem scenarios where the client was correctly sending and receiving emails, and then for the faulty scenarios listed in Table 3. We reproduced each of the problems listed in the table on each OS by manually configuring the email client to reproduce the problem. We collected 5 samples for each of the problems on each OS. Thus, we collected a total of 15 samples for each problem. In all, we collected 150 traces of email problems: 30 GOOD and 120 BAD.

Note that we cleared the DNS cache before each run to capture the complete network activity of the email client. Had we had not done so, the small size of our experimental setup (3 clients and 2 servers) would have meant that DNS queries would have been largely absent, having been filtered out by the cache. However, in a real setting, with a large number of clients and servers, there would be DNS queries associated with at least a fraction of the successful transactions. We seek to recreate such instances despite the small size of our setup, by clearing the DNS cache before each run.

Features: The university uses IMAP and SMTP over SSL, hence, the feature extractor summarizes each email trace using features specific to SMTP over SSL, IMAP over SSL, TCP on the respective ports, DNS (this is the only name resolution protocol in use in the university network), and generic features that can help capture low-level problems, such as the presence or absence of background ARP and IP packets. Since all traces involve the client machines connecting to one of two mail servers, we also record aggregate features, as discussed in Section 3, to capture server-specific behavior.

5.2 Comparison with a Classifier

Our first evaluation concentrates on comparing the signatures learned by Deja vu with those learned by a conventional classifier. While we input traces labeled as either GOOD or BAD to Deja vu, the classifier had the added benefit of having each faulty trace labeled with the root cause of the problem instead of just the generic BAD label. To perform classification, we used the C4.5 decision tree classifier, which has been used often in prior work [2, 5].

Problem	**Root Cause**	**Configuration Details**	**# of Traces Collected**
Internal corporate sites fail to load with opera.	Unsupported authentication protocol.	Opera under certain Oses fails to perform NTLM authentication correctly.	13 bad traces with Opera only.
Internal corporate sites unreachable by any browser.	Wrong browser configuration.	The browser tries to use the proxy to reach internal sites.	60 bad traces.
Certain websites display error "Forbidden: You dont have permission to access this server." But accessing them via different proxies loads the website fine.	Certain corporate proxies are blocked by these websites, possibly due to excessive requests. Accessing the websites via these blocked proxies displays the error.	e.g. Yelp.com had blocked some subset of proxies in our setting. But accessing Yelp.com via other proxies worked.	64 traces via good proxies, and 64 via blocked proxies.
Certain websites silently fail to load; no error is displayed to the user.	Flash or Ad blockers installed in the browser prevent loading some components, which are critical for loading these sites.	e.g. Pandora.com silently fails to load in Firefox when Flashblock 1.5.13 add-on is enabled.	92 traces with flash blocking and 92 without blocking.
Websites with popular third-party scripts fail to load in IE, making these websites unusable.	IEs InPrivate Filtering, when enabled, blocks loading of third-party scripts that are commonly found in websites.	Components of the websites built using scripts such as Google Analytics, or recaptcha.net fail to load.	10 good and 10 bad traces only with IE.
Internal corporate sites fail to load except in IE8.	VBScript not supported in all browsers except IE8.	Sites used to manage internal information heavily use VBScript.	10 good and 33 bad traces.
Some websites fail to load silently.	Wrong HTTPS proxy server configuration	Pandora.com fails to load some flash component, which fails the entire website.	55 bad traces.
None of the websites load in the browser.	Firewall on the client is blocking web browsing.	Firewall blocks port 80.	64 good and 64 bad traces.
None of the websites load in the browser.	Wrong proxy configuration.	Configured a wrong proxy server in the browser.	15 good and 64 bad traces.
Some websites fail to load.	User types in a wrong URL into the address bar.	Entered wrong URL into the browser.	64 bad traces.
All websites fail to load.	Wrong DNS server configuration.	Manually configured wrong DNS server address.	52 good and 52 bad traces in Win 7 only.

Table 2: **Browser trace details.**

Figure 2 and Figure 3 show the root causes for our email and browser datasets, and the signatures that Deja vu and the classifier learned for each root cause (depicted through arrows pointing out from the root cause boxes). The labels on the side of each signature give the serial number with which we refer to the signature in this section, followed by the number of BAD traces that contributed to learning that signature. Note that one root cause could have multiple signatures; for instance, an incorrect proxy setting could lead to distinct signatures with different browsers. Likewise, multiple root causes could share the same signature; for instance, the absence of SYNACKs could be because of a wrong server address being used or a wrong port number.

We do not expect the reader to parse each signature in detail. The graphics are only meant to convey the similarities and differences between the two signature sets, which we touch on through specific examples in Section 5.2.1. Similar parts of Deja vu's signatures and the corresponding classifier signatures are in bold. Note that the browser signatures are more detailed (involving a larger number of features) and also more diverse (one-to-many mapping between root causes and signatures) than the email signatures. This is so for both Deja vu and the classifier, partly because the browser data set contained data from 5 different browsers.

Equivalence: To measure how equivalent the signatures learned by Deja vu and the classifier are, we compute a *difference metric* between the signature sets. The intuition behind this metric is that even if a Deja vu signature and the corresponding classifier signature look very different, these might still be equivalent depending on how they categorize the traces. For every pair of BAD traces in the training set, we check to see if both traces share a Deja vu signature and, separately, whether they share a classifier signature. If a pair of traces shares a signature in both cases or does not in either case, that means that both the Deja vu signatures and the classifier signature are equivalent in terms of how they categorize the two traces. However, if the signature is shared in one case but not in the other, there is a mismatch and so we increment the difference metric. Finally, we normalize the metric by total number of faulty trace pairs.

For the email traces, there were 120 BAD traces, of which we consider a total of 6329 trace pairs, since some of the traces were deemed as noisy by either Deja vu or the classifier, and did not contribute to a signature. Of these 6329 pairs, only 189 differed in the sense we have described above, yielding a normalized difference metric of 3%. For the browser dataset, there were a total of 307 BAD traces, of which we consider 40186 pairs. Of these, only 1806 differed, giving us a normalized difference metric of 4.5%. Thus, despite operating with just coarse-grained GOOD/BAD labels, Deja vu is able to

Problem	**Root Cause**
Cannot view email. Client returns error message "login to server <server-name> failed."	Wrong username configured in the email client.
Cannot connect to the incoming email server. Client returns error "Failed to connect to server <email-address>."	Wrong incoming server name configured in the client.
Cannot receive emails. The client returns error "The IMAP server <server-name> does not support the selected authentication method. Please change the 'Authentication method' in the 'Account Settings | Server Settings'."	Wrong authentication mode configured with the incoming server in the client.
Cannot receive emails. Client returns error "Could not connect to the server <server-name>; the connection was refused."	Wrong incoming server port number configured.
Cannot send emails. The client returns error "Sending of message failed. The SMTP server <server-name> does not support the selected authentication method. Please change the 'Authentication method' in the 'Account Settings | Outgoing Server (SMTP)'."	Wrong authentication mode configured with the outgoing server in the email client.
Cannot send emails. Client returns "Sending of message failed. An error occurred sending mail: SMTP server <server-name> is unknown. The server may be incorrectly configured. Please verify that your SMTP server settings are correct and try again."	Wrong outgoing server name configured in the email client.
Cannot send emails. Client returns "Sending of message failed. The message could not be sent because connecting to SMTP server <server-name> failed. The server may be unavailable or is refusing connections. Please verify that your server settings are correct and try again, or contact the administrator."	Wrong outgoing server port number configured in the email client.
Sending a message while Thunderbird is in "Offline Mode" puts the message in "outgoing" folder rather than sending it.	User has forgotten that "offline mode" is enabled.

Table 3: Email trace details. We collected 15 traces for each problem (5 per OS).

learn signatures almost as effectively as a classifier that has access to fine-grained labels corresponding to the root causes.

5.2.1 Signature comparison

We now subjectively compare the Deja vu signatures to the classifier signatures.

1. *Sometimes, Deja vu signatures provide more information than classifier signatures.* All three signatures – D3, D4, and D5 in Figure 3 – capture the fact that there is no successful Netbios response in the traces (NBTNSRS = 0), whereas C2 does not. In fact, the Deja vu signatures capture the actual root cause, because it is *only* when both Netbios and DNS fail, that name resolution fails in the corporate network. The classifier signature does not capture this because capturing only the DNS failure is enough to differentiate this failure category from other failure categories. This example shows the benefits of using a learning approach like Deja vu that compares BAD traces with all the GOOD traces at every step.

Another example is signature D12 in Figure 3. We investigated why there is no corresponding signature with the classifier, and found that the D12 signature specifically captures behavior of the Firefox and Opera browsers, which behave differently under the "Wrong URL" root cause compared to other browsers. The classifier does not capture this behavior because its input labels are only at the granularity of the root cause, and does not distinguish between browsers [1].

[1]We also tried inputting more fine-grained labels of the form OS:Browser:RootCause to the classifier. This gave us very noisy signatures since the classifier was forced to choose spurious features to differentiate between similar traces from different browsers and OSes.

2. *Sometimes, Deja vu signatures did not differentiate root causes but the classifier signatures did.* Figure 2 shows that Deja vu did not pick out the right feature differentiating root causes "Wrong Outgoing Port But Correct Mail Server" (which would involve a successful DNS resolution of the mail server name) and "Wrong Outgoing Mail Server" (which would not include a successful DNS resolution because the wrong mail server name does not correspond to any real host). The reason Deja vu did not pick this feature is that in 6 of the 15 BAD traces for the former root cause there was background DNS traffic that was failing, which confused Deja vu into believing that such DNS lookup failure was distributed across both of the above root causes, and therefore not useful for separating them. In fact, this confused the classifier too, which attributed the signature for "Wrong Outgoing Mail Server" to the former root cause also. Removal of such background noise can aid Deja vu's learning significantly, as noted in Section 7.

3. *Deja vu signatures are in general longer than the classifier signatures.* For example, in Figure 2, signature D1 (the Deja vu signature for the "wrong incoming mail server address" root cause) is 6 features long, whereas the corresponding classifier signature C1 is only 3 features long. The reason for the longer signatures is that, at each iteration, Deja vu can choose a feature only to differentiate BAD traces from GOOD ones, whereas the classifier has the added freedom of choosing a feature that directly differentiates between different kinds of BAD traces. On the flip side, however, the Deja vu signatures provide more insight into the failure. For example, signature D1 from Deja vu tells us that there was no successful TCP handshake on port

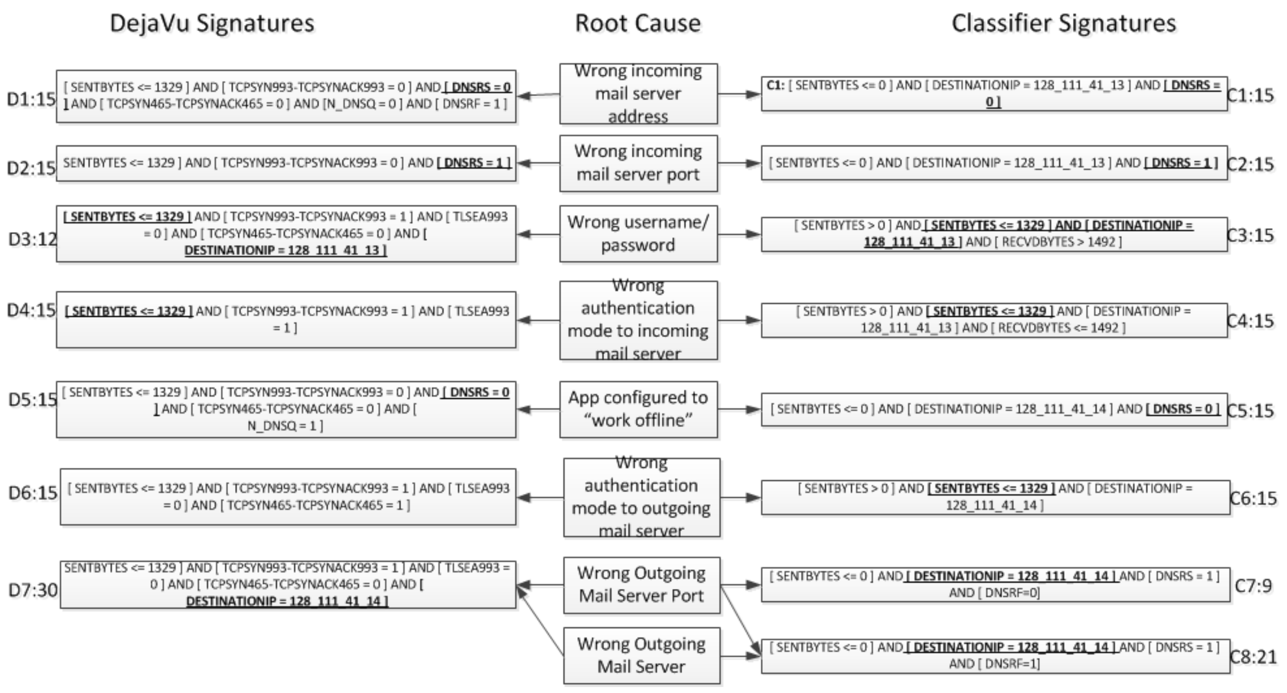
DejaVu Signatures
Root Cause
Classifier Signatures
D1:15
[SENTBYTES <= 1329] AND [TCPSYN993-TCPSYNACK993 = 0] AND [DNSRS = 0] AND [TCPSYN465-TCPSYNACK465 = 0] AND [N_DNSQ = 0] AND [DNSRF = 1]
Wrong incoming mail server address
C1: [SENTBYTES <= 0] AND [DESTINATIONIP = 128_111_41_13] AND [DNSRS = 0]
C1:15
D2:15
SENTBYTES <= 1329] AND [TCPSYN993-TCPSYNACK993 = 0] AND [DNSRS = 1]
Wrong incoming mail server port
[SENTBYTES <= 0] AND [DESTINATIONIP = 128_111_41_13] AND [DNSRS = 1]
C2:15
D3:12
[SENTBYTES <= 1329] AND [TCPSYN993-TCPSYNACK993 = 1] AND [TLSEA993 = 0] AND [TCPSYN465-TCPSYNACK465 = 0] AND [DESTINATIONIP = 128_111_41_13]
Wrong username/ password
[SENTBYTES > 0] AND [SENTBYTES <= 1329] AND [DESTINATIONIP = 128_111_41_13] AND [RECVDBYTES > 1492]
C3:15
D4:15
[SENTBYTES <= 1329] AND [TCPSYN993-TCPSYNACK993 = 1] AND [TLSEA993 = 1]
Wrong authentication mode to incoming mail server
[SENTBYTES > 0] AND [SENTBYTES <= 1329] AND [DESTINATIONIP = 128_111_41_13] AND [RECVDBYTES <= 1492]
C4:15
D5:15
[SENTBYTES <= 1329] AND [TCPSYN993-TCPSYNACK993 = 0] AND [DNSRS = 0] AND [TCPSYN465-TCPSYNACK465 = 0] AND [N_DNSQ = 1]
App configured to “work offline”
[SENTBYTES <= 0] AND [DESTINATIONIP = 128_111_41_14] AND [DNSRS = 0]
C5:15
D6:15
[SENTBYTES <= 1329] AND [TCPSYN993-TCPSYNACK993 = 1] AND [TLSEA993 = 0] AND [TCPSYN465-TCPSYNACK465 = 1]
Wrong authentication mode to outgoing mail server
[SENTBYTES > 0] AND [SENTBYTES <= 1329] AND [DESTINATIONIP = 128_111_41_14]
C6:15
D7:30
SENTBYTES <= 1329] AND [TCPSYN993-TCPSYNACK993 = 1] AND [TLSEA993 = 0] AND [TCPSYN465-TCPSYNACK465 = 0] AND [DESTINATIONIP = 128_111_41_14]
Wrong Outgoing Mail Server Port
[SENTBYTES <= 0] AND [DESTINATIONIP = 128_111_41_14] AND [DNSRS = 1] AND [DNSRF=0]
C7:9
Wrong Outgoing Mail Server
[SENTBYTES <= 0] AND [DESTINATIONIP = 128_111_41_14] AND [DNSRS = 1] AND [DNSRF=1]
C8:21

Figure 2: **Root causes and corresponding Deja vu signatures and classifier signatures for email traces.**

993 (IMAP over SSL) [TCPSYN993-TCPSYNACK993 = 0] whereas the classifier signature does not give us this information.

4. Sometimes, Deja vu has multiple signatures corresponding to the same root cause, whereas the classifier does not. Since Deja vu does not have access to fine-grained labels, it sometimes creates noisy splits in the signature which the classifier avoids. Such noisy splits can be avoided to some extent by techniques noted in Section 7.

5.3 Signature Stability and Adaptability

Next, we turn to the question of how stable Deja vu's signatures are, and how effective the algorithm is at learning new signatures on-the-fly. Would the signatures learned from a training set still apply to a test set gathered at a later time? Does the algorithm learn new signatures when required?

To answer these questions, we collected a test dataset in the corporate network for two browsers – IE and Firefox – approximately 2 months after we had collected the training dataset. We collected training data on Windows 7, Mac OSX, and Ubuntu systems, and the test dataset on a Windows XP machine. The test dataset included 10 BAD traces (5 each for IE and Firefox) for each of 6 root causes, giving us a total of 60 bad traces. We could not collect data for the "Misconfigured outgoing firewall" root cause because Windows XP does not allow the configuration of outgoing firewall rules.

For 5 out of the 6 root causes, an overwhelming majority (95%) of the traces in the test dataset matched the signatures of the same root cause that had been learnt earlier from the training dataset. This demonstrates the stability of the Deja vu signatures for these 5 root causes. However, Deja vu misclassified *all* 10 traces for the "Wrong proxy" root cause, marking them either as "internal site authentication error" or "name resolution error".

To investigate this, we relearned the Deja vu signatures by adding these 10 BAD traces to the initial training set of 878 traces. We found that Deja vu learned an additional, new signature for the "Wrong proxy" root cause, which all 10 new traces contributed to:

$[HTTPR200 = 0]$ AND $[N\_HTTPQ = 0]$ AND $[HTTPR502 = 0]$ AND $[HTTPR500 = 0]$ AND $[HTTPR504 = 1]$

To create the "Wrong proxy" root cause, we always set the proxy to a non-existent IP address that we were confident would not respond (e.g., 5.1.1.1). However, the new signature noted above indicates that not only did requests to this IP address complete a successful TCP handshake, it even responded with an HTTP Gateway error! We communicated this to the relevant network administrators, who investigated the matter and then informed us that this strange behavior was the result of some recent routing configuration changes made on the corporate network that directed traffic to some non-existent IP addresses to a set of misconfigured servers that were responding to the requests with a gateway error.

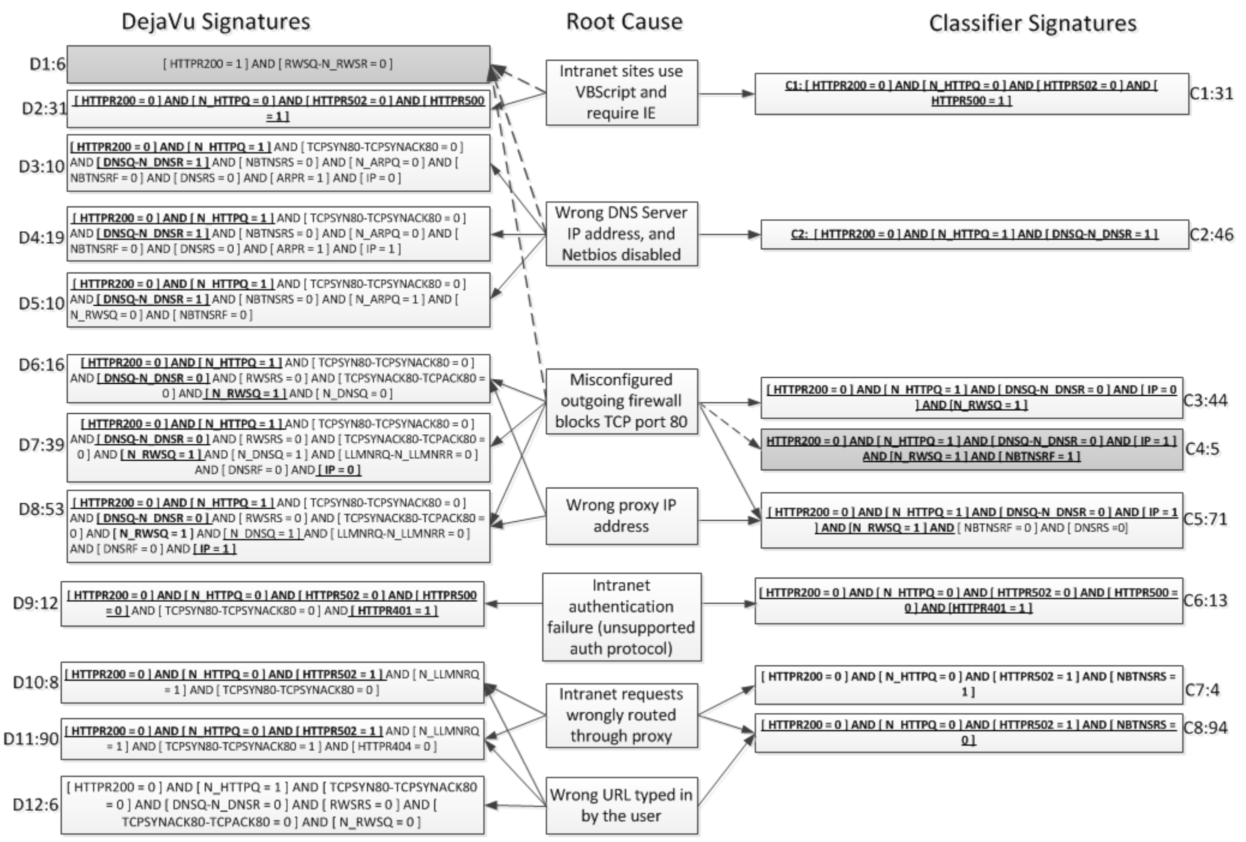
DejaVu Signatures
Root Cause
Classifier Signatures
D1:6
[HTTPR200 = 1] AND [RWSQ-N_RWSR = 0]
D2:31
[HTTPR200 = 0] AND [N_HTTPQ = 0] AND [HTTPR502 = 0] AND [HTTPR500 = 1]
D3:10
[HTTPR200 = 0] AND [N_HTTPQ = 1] AND [TCPSYN80-TCPSYNACK80 = 0] AND [DNSQ-N_DNSR = 1] AND [NBTNSRS = 0] AND [N_ARPQ = 0] AND [NBTNSRF = 0] AND [DNSRS = 0] AND [ARPR = 1] AND [IP = 0]
D4:19
[HTTPR200 = 0] AND [N_HTTPQ = 1] AND [TCPSYN80-TCPSYNACK80 = 0] AND [DNSQ-N_DNSR = 1] AND [NBTNSRS = 0] AND [N_ARPQ = 0] AND [NBTNSRF = 0] AND [DNSRS = 0] AND [ARPR = 1] AND [IP = 1]
D5:10
[HTTPR200 = 0] AND [N_HTTPQ = 1] AND [TCPSYN80-TCPSYNACK80 = 0] AND [DNSQ-N_DNSR = 1] AND [NBTNSRS = 0] AND [N_ARPQ = 1] AND [N_RWSQ = 0] AND [NBTNSRF = 0]
D6:16
[HTTPR200 = 0] AND [N_HTTPQ = 1] AND [TCPSYN80-TCPSYNACK80 = 0] AND [DNSQ-N_DNSR = 0] AND [RWSRS = 0] AND [TCPSYNACK80-TCPACK80 = 0] AND [N_RWSQ = 1] AND [N_DNSQ = 0]
D7:39
[HTTPR200 = 0] AND [N_HTTPQ = 1] AND [TCPSYN80-TCPSYNACK80 = 0] AND [DNSQ-N_DNSR = 0] AND [RWSRS = 0] AND [TCPSYNACK80-TCPACK80 = 0] AND [N_RWSQ = 1] AND [N_DNSQ = 1] AND [LLMNRQ-N_LLMNRR = 0] AND [DNSRF = 0] AND [IP = 0]
D8:53
[HTTPR200 = 0] AND [N_HTTPQ = 1] AND [TCPSYN80-TCPSYNACK80 = 0] AND [DNSQ-N_DNSR = 0] AND [RWSRS = 0] AND [TCPSYNACK80-TCPACK80 = 0] AND [N_RWSQ = 1] AND [N_DNSQ = 1] AND [LLMNRQ-N_LLMNRR = 0] AND [DNSRF = 0] AND [IP = 1]
D9:12
[HTTPR200 = 0] AND [N_HTTPQ = 0] AND [HTTPR502 = 0] AND [HTTPR500 = 0] AND [TCPSYN80-TCPSYNACK80 = 0] AND [HTTPR401 = 1]
D10:8
[HTTPR200 = 0] AND [N_HTTPQ = 0] AND [HTTPR502 = 1] AND [N_LLMNRQ = 1] AND [TCPSYN80-TCPSYNACK80 = 0]
D11:90
[HTTPR200 = 0] AND [N_HTTPQ = 0] AND [HTTPR502 = 1] AND [N_LLMNRQ = 1] AND [TCPSYN80-TCPSYNACK80 = 1] AND [HTTPR404 = 0]
D12:6
[HTTPR200 = 0] AND [N_HTTPQ = 1] AND [TCPSYN80-TCPSYNACK80 = 0] AND [DNSQ-N_DNSR = 0] AND [RWSRS = 0] AND [TCPSYNACK80-TCPACK80 = 0] AND [N_RWSQ = 0]
Intranet sites use VBScript and require IE
Wrong DNS Server IP address, and Netbios disabled
Misconfigured outgoing firewall blocks TCP port 80
Wrong proxy IP address
Intranet authentication failure (unsupported auth protocol)
Intranet requests wrongly routed through proxy
Wrong URL typed in by the user
C1: [HTTPR200 = 0] AND [N_HTTPQ = 0] AND [HTTPR502 = 0] AND [HTTPR500 = 1]
C1:31
C2: [HTTPR200 = 0] AND [N_HTTPQ = 1] AND [DNSQ-N_DNSR = 1]
C2:46
[HTTPR200 = 0] AND [N_HTTPQ = 1] AND [DNSQ-N_DNSR = 0] AND [IP = 0] AND [N_RWSQ = 1]
C3:44
HTTPR200 = 0] AND [N_HTTPQ = 1] AND [DNSQ-N_DNSR = 0] AND [IP = 1] AND [N_RWSQ = 1] AND [NBTNSRF = 1]
C4:5
[HTTPR200 = 0] AND [N_HTTPQ = 1] AND [DNSQ-N_DNSR = 0] AND [IP = 1] AND [N_RWSQ = 1] AND [NBTNSRF = 0] AND [DNSRS =0]
C5:71
[HTTPR200 = 0] AND [N_HTTPQ = 0] AND [HTTPR502 = 0] AND [HTTPR500 = 0] AND [HTTPR401 = 1]
C6:13
[HTTPR200 = 0] AND [N_HTTPQ = 0] AND [HTTPR502 = 1] AND [NBTNSRS = 1]
C7:4
[HTTPR200 = 0] AND [N_HTTPQ = 0] AND [HTTPR502 = 1] AND [NBTNSRS = 0]
C8:94

Figure 3: Root causes and corresponding Deja vu signatures and classifier signatures for browser traces.

This interesting anecdote shows that Deja vu signatures are not just useful for failure classification, but can also be a component of a *network problem diagnosis tool*. Whenever Deja vu learns a new problem signature, the tool can alert the administrators so they can investigate it to see if the signature reveals anything untoward.

6. APPLICATIONS

Deja vu provides a way to characterize network problems with a compact fingerprint. Such a fingerprint has several applications. A fingerprint could be used to *search* through a large dataset to find instances of a particular problem. It could also be used to *recall* and *match* against previously seen instances of a problem. We briefly describe applications in each of these categories.

6.1 Search Tool

Packet tracing tools such as `tcpdump` and `netmon` provide a way to apply filters to find specific packet types of interest, either from a live capture or from a recorded trace. However, what if we are interested in searching for *problem events* rather than for specific packets? For instance, we might wish to find instances in a trace where a secure webpage access failed because the firewall blocked port 443 traffic. To provide this capability, we have built a simple search tool using Deja vu. The target trace is sliced into windows, either sliding windows or jumping windows. Features extracted from each slice are then fed into the signature tree constructed by Deja vu for the problem of interest.

One question is how wide a slice should be. Ideally, the slice should be wide enough to accommodate the problem event of interest but no wider. For instance, consider a problem signature that comprises a successful DNS request-response exchanged followed by a successful TCP SYN handshake followed, in turn, by an HTTP request that fails to elicit a response. To be able to capture the full signature, the slice must be wide enough to span all of the above packet exchanges. However, if the slice were too wide, then it risks being polluted by noise in the form of features from packets belonging to an unrelated transaction.

In our implementation of the search tool, we use a slice size of 30 seconds. We tested the tool on a 4MB network trace collected over a period of 40 minutes. We recreated 5 different problems from the set of root causes shown in Figure 3. The search tool was successful in finding 3 of these problems. It missed catching one problem because the window size was too large and background noise polluted it, and it missed the second one because the window size was too small to capture an important feature later in the trace. This indicates that such a search tool should ideally use windows of varying sizes to catch all problems in the trace.

6.2 Help Desk Application

A second application of Deja vu is in the context of a help desk tool. Our help desk application uses the problem signatures generated by Deja vu to automatically match the problem being experienced by the user against a database of known issues, i.e., ones for which there is a known fix. Whenever a failure is encountered (e.g., a browser error), the Deja vu component on the client machine extracts features from the packet trace in the recent past (tracing is an ongoing background activity) and sends these to the Deja vu server. At the server, these features are fed into the application's signature tree and thereby matched against a known category of failures. The problem notes associated with this category would then guide the diagnosis and resolution procedure.

A more sophisticated version of the help desk application could use Deja vu signatures to index WikiDo [9] tasks instead of just indexing manually crafted notes.

7. DISCUSSION

Noisy traces: We discuss the impact of noisy traces on Deja vu's signatures. Noise refers to packets that are extraneous to the application of interest. Such noise could arise from the network communication of other applications or even other hosts, depending on where the packet trace is captured. Deja vu's feature extractor would then extract features from such background traffic and include these with the (correct) features corresponding to the traffic of interest.

Such noisy features could be problematic in two ways: (a) these could lead Deja vu to learn incorrect signatures for problems, and (b) these could cause an incorrect match when an attempt is made to match the noisy features against the signatures generated by Deja vu.

Deja vu's use of GOOD/BAD labels helps mitigate problem (a) because the noisy features are likely to be uncorrelated with the success (GOOD) or failure (BAD) of the application of interest and hence are likely to be disregarded by Deja vu's signature construction algorithm. However, a noisy feature extracted from background traffic (e.g., a successful DNS request-response exchange) could still cause problems, as explained above.

To alleviate the above problem, we could leverage prior work on application traffic fingerprinting (e.g., [11, 7, 15] to separate out just the subset of traffic in a packet trace that corresponds to the application of interest. Performing such separation thoroughly would require the tracing to be performed on the end hosts, so that traffic could be unambiguously tied to specific applications.

Another source of inaccuracy in the traces is mislabeling of GOOD and BAD traces. Previous work [2] has shown that the C4.5 decision tree algorithm is robust to a certain degree of mislabeling in the context of network diagnostics. However, no learning algorithm can withstand large amounts of mislabeling. Applications that use Deja vu have to be designed in a way so that the chances of mislabeling stays low. A discussion of such application-level techniques are out of scope of this paper.

Scalability: In our experiments, the Deja vu algorithm took less than one second to complete processing all the traces. For the applications we have discussed, we expect practitioners to run Deja vu with a frequency of approximately once a day, and we believe the current performance is suitable for this design point. It is, however, possible that as the problem traces become more diverse, Deja vu may learn a considerable number of problem signatures in a single run. In such cases, signatures can be prioritized based on the confidence that the C4.5 algorithm assigns onto them. Signatures that are seen more often can be bubbled to the top of the priority list, thereby allowing an administrator or support engineer to look at the more predominant problems first.

8. RELATED WORK

8.1 Network Traffic Analysis

Analysis of network traffic has been used to fingerprint applications and infer the behavior of protocols [15, 11]. While such analysis has used supervised learning on coarse features such as packet size and flow length to distinguish between applications, Deja vu operates on more fine-grained features (e.g., features specific to DNS, TCP, HTTP, etc.) but with coarse-grained GOOD vs. BAD labels.

Such analysis has also been used to discover the session-level structure of applications [7], e.g., to discover that in an FTP session, a control connection is often followed by one or more data connections. However, to our knowledge, such session structure has not been used for constructing signatures for network problems. Furthermore, discovering session structure is only semi-automated, requiring the involvement of a human expert to actually reconstruct the session structure. Human involvement in Deja vu is limited to labeling training runs as GOOD or BAD, a much less onerous task.

Finally, such analysis has also been used to perform network anomaly detection (e.g., [18]). The typical approach has been to construct a model of normal behaviors based on past traffic history and then look for significant changes in short-term behavior based that are inconsistent with the model. While anomaly detection has focused on aggregate behavior, Deja vu focuses on the network behavior of an individual application run.

8.2 Fingerprinting Problems

DebugAdvisor [3] is a tool to search through source control systems and bug databases to aid debugging. Unlike Deja vu, it uses a standard text search tool over call stack information and bug reports. Deja vu is closer in spirit to work on automating the diagnosis of system problems, which involves extracting signatures from information such as system call traces (e.g., [16]). The approach is to employ supervised learning (e.g., SVM) on a fully labeled database of known problems. In a similar vein, Clarify [5] is a system that improves error reporting by classifying application behavior. Clarify generate a behavior profile, i.e., a summary of the program's execution history, which is then labeled by a human expert to enable learning-based classification.

In comparison with the above approaches, which require a human expert to perform full labeling, Deja vu operates only with coarse-grained labels. Also, since Deja vu focuses on network problems, there are a number of domain-specific choices it incorporates, including for feature selection.

8.3 State-based Diagnosis

STRIDER [14] and PeerPressure [13] analyze state information in the Windows registry, to identify features (e.g., registry key settings) that are indicative of a problem. Unlike with Deja vu, the goal of this body of work was not to develop problem-specific signatures based on the behavior of the system. Rather it is to detect anomalous state by performing state differencing between a health machine and a sick machine. Also, the features (e.g., registry key settings) were treated as opaque entities whereas Deja vu uses networking domain-specific knowledge to define features.

Similarly, NetPrints [2] analyzes network configuration information to diagnose home network problems. While being largely state-based, NetPrints also made limited use of network problem signatures to address the issue of hidden configurations that are not available to the state-based analysis.

Compared to the above, Deja vu is not intrusive since it operates on network traffic and hence does not require any tracing to be performed on the end system itself.

8.4 Active Probing

While Deja vu seeks to extract network problem signatures from existing application traffic, there is a large body of work on characterizing network problems through active probing [10, 17, 4, 8]. Active probing with a carefully-crafted set of tests enables detailed characterization of a range of problems, often enabling diagnosis. In contrast, Deja vu strives to produce a problem fingerprint based on the traffic that the application generates anyway. These fingerprints may not contain the detail to directly enable diagnostics. Nevertheless, these provide a generic way to match a problem instance with a previously seen instance, thereby enabling diagnostics, as noted in Section 6.2.

9. CONCLUSION

Deja vu is a tool to associate a compact signature with each category of network problem experienced by an application. It uses a novel algorithm to learn the signatures from coarse-grained GOOD/BAD labels. Our experimental evaluation, including comparison with a standard classifier (which has the benefit of knowing fine-grained labels) and a user study, has demonstrated the effectiveness of Deja vu signatures.

10. REFERENCES

[1] Microsoft network monitor. URL "http://www.microsoft.com/downloads/en/netmon".

[2] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. Padmanabhan, and G. Voelker. NetPrints: Diagnosing Home Network Misconfigurations using Shared Knowledge. In *NSDI*, 2009.

[3] B. Ashok, J. Joy, H. Liang, S. Rajamani, G. Srinivasa, and V. Vangala. DebugAdvisor: A Recommender System for Debugging. In *FSE*, 2009.

[4] M. Dischinger, M. Marcon, S. Guha, K. P. Gummadi, R. Mahajan, and S. Saroiu. Glasnost: Enabling End Users to Detect Traffic Differentiation. In *Networked Systems Design and Implementation*, 2010.

[5] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved Error Reporting for Software that Uses Black-box Components. In *PLDI*, 2007.

[6] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, and J. Padhye. Detailed diagnosis in computer networks. In *Sigcomm*. ACM, 2010.

[7] J. Kannan, J. Jung, V. Paxson, and C. E. Koksal. Semi-Automated Discovery of Application Session Structure. In *IMC*, 2006.

[8] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating The Edge Network. In *IMC*, 2010.

[9] N. Kushman, M. Brodsky, S. Branavan, D. Katabi, R. Barzilay, and M. Rinard. WikiDo. In *HotNets*, 2009.

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

[11] A. Moore and K. Papagiannaki. Toward the Accurate Identification of Network Applications. In *PAM*, 2005.

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

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

[14] 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.

[15] C. V. Wright, F. Monrose, and G. M. Masson. On Inferring Application Protocol Behaviors in Encrypted Network Traffic. *J. Machine Learning Research*, Dec 2006.

[16] 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.

[17] Y. Zhang, Z. M. Mao, and M. Zhang. Effective Diagnosis of Routing Disruptions from End Systems. In *Networked Systems Design and Implementation*, 2008.

[18] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund. Sketch-based Change Detection: Methods, Evaluation, and Applications. In *IMC*, 2004.