Rippler: Delay Injection for Service Dependency Detection

Ali Zand, Giovanni Vigna, Richard Kemmerer, and Christopher Kruegel

ABSTRACT

Detecting dependencies among network services has been well-studied in previous research. Previous attempts at service dependency detection fall into two classes: active and passive approaches. While passive approaches suffer from high false positives, active approaches suffer from applicability problems.

In this paper, we design a new application-independent active approach for detecting dependencies among services. We present a traffic watermarking approach with arbitrarily low false positives and easy applicability. Our approach does not need any modifications to or implementation details about the existing network services. We provide ways to watermark sets of network flows and later detect these watermarks dependably. We provide statistical tests for detecting watermarked flows, and we compute the false positive and false negative rates of these tests both analytically and experimentally. We also provide a set of criteria for the evaluation of dependency detection approaches and compare our approach to previous ones using these criteria.

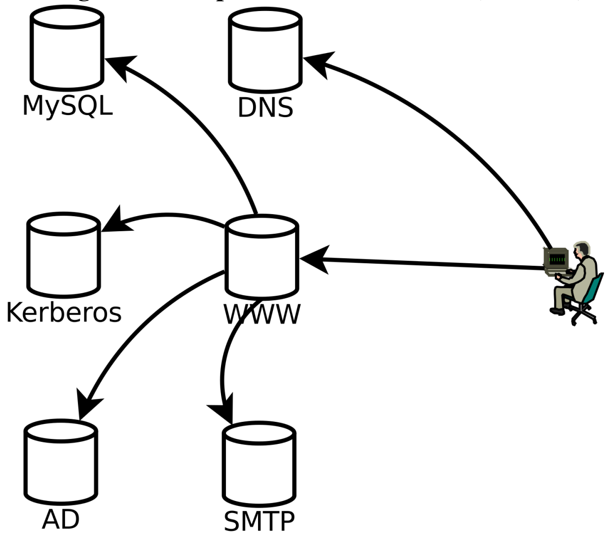
Furthermore, we implemented the proposed watermarking system (*Rippler*) in a small university lab network. We ran our system for four months and detected 38 otherwise hidden dependencies among 54 services. Finally, we compared the efficiency of our approach against three previous systems by testing them on this real-world network data.

1. INTRODUCTION

Corporate and governmental computer networks are targets of constant attacks [16]. Although the attackers targeting these networks may have different incentives, goals, and techniques, they generally target the same aspects of the system: confidentiality, integrity, and availability. While confidentiality and integrity properties have historically attracted more attention from the security community, availability property has been comparatively neglected.

We depend on network services for many of our daily needs (e.g., Internet banking, personal accounting, social networking, and medical services). The ubiquity and diversity of network services have led to an ever-increasing complexity of the infrastructure supporting these services. As engineers use divide-and-conquer to attack complexity, these services are implemented as composite modules, built of multiple, simpler, underlying services. This modular approach enables designers to reuse standard services to build complex customized ones. For example, a webmail service is usually implemented using several simple modules including a web service, an email service, and a DNS service.

Figure 1: Sample distributed service (webmail)



This modular design paradigm has security and reliability implications. On the one hand, the modular design along with reusing and sharing modules makes it challenging to determine a distributed system's perimeters. This, in turn, can lead to insecure network topology design. On the other hand, the modular design makes it challenging to prioritize security events and assets, to correlate security alerts, to generate attack graphs, and to provide situational awareness.

As services become more complex and increasingly distributed, protecting them becomes more challenging. Because there are more components that can fail and make the whole service unavailable, distributed systems are generally more difficult to protect. One needs to know the components of a composite service to be able to protect it. Unfortunately, these implementation and dependency details are often undocumented and difficult to identify in complex networks.

Figure 1 shows a composite webmail service. A typical client checks her email using a web interface, by first contacting a DNS server to acquire the IP address of the webserver. The webserver, in turn, contacts a Kerberos server to authenticate the user, an Active Directory server to load the user's contact list, a MySQL server to load the user's profile, and an SMTP server to send the user's email. If any of the involved services fail, the final webmail service will fail or will be degraded. The system administrator needs to know the dependencies between the involved services to be able to adequately protect the webmail service.

Previous work on service dependency detection can be divided into active [10] and passive [13] approaches. Passive approaches do not generate any additional traffic. They simply observe the existing traffic and find the set of services that exhibit correlated activity. Active approaches, on the other hand, manipulate the timing or the contents of the traffic to identify dependencies. Each of these approaches has its own advantages and disadvantages.

Passive approaches suffer from two main problems: higher false positive rates and the inability to detect the direction of the dependency relations (who depends on whom). These problems result from the fact that "correlation does not imply causation." In other words, when two services are correlated with each other it does not necessarily mean that they depend on each other. For example, two services may depend on and be influenced by a third service, and that is the reason why their activities are correlated. Noise and jitters in a real-world network can also cause occasional spurious correlated activity in the services. This problem leads to the detection of false dependencies (false positives). Even when one service depends on a second one, the correlation does not show which service depends on the other.

Active approaches are harder to apply, as they require higher level of access to the individual systems, require more modifications to the system, and are usually application dependent, and they even may introduce more load into the network (e.g., by adding tags to application-specific traffic). The high level of access and high level of modification to the network that are required by active approaches make their application in a real production system, at the very least, challenging. Moreover, application-dependent approaches cannot be used for detecting dependencies between unknown types of services.

The main cause of high false positives in passive approaches is their inability to differentiate between correlation and causation. In active approaches, one can differentiate between correlation and causation, by controlling one variable and observing the other. This process is not possible with passive methods. Therefore, active approaches are needed to recognize the direction of dependencies, and to mitigate the false positive problem.

In this paper, we provide an active watermarking approach that is application-independent and inflicts minimal burden on the network. To detect dependencies, we create temporal perturbation patterns in request arrival timings for different services, and we determine whether or not these patterns propagate to other services. We provide an analytical framework to interpret the results of the experiments using statistical inference. More specifically, we use three different statistical tests to show the existence of the dependency relationship. We analytically show that any desirable level of accuracy can be achieved if the experiment running time is long enough. We also validate this high accuracy by simulation. We implemented a watermarking system, called *Rippler*. Finally, we deployed this watermarking system in a university computer laboratory network and detected 38 dependencies.

Our approach requires the ability to selectively delay packets and also access to a network dataset that contains the information about each individual network connection start and end time. Any frequently used network traffic dump format, such as NetFlow [7] records or tcpdump, contains the required information about the network connections.

Our contributions are the following:

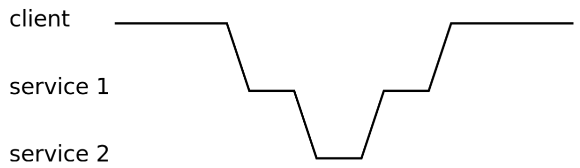
- We provide a novel application-independent flow-watermarking approach for detecting service dependencies.
- We provide statistical models of the watermarking approach and provide three statistical tests for detecting service dependencies. We show that the suggested tests can achieve arbitrarily small error probabilities given large-enough data samples.
- We implemented a flow-watermarking system, and we installed it in a university department network and analyzed the gathered data. Our system detected 38 dependencies using this system, some of which were not previously identified by the system administrators.
- We compared the results of our approach to three previous works in dependency detection. We showed that *Rippler* outperformed the passive approaches, and produced outputs with high levels of confidence.

2. SERVICE DEPENDENCY

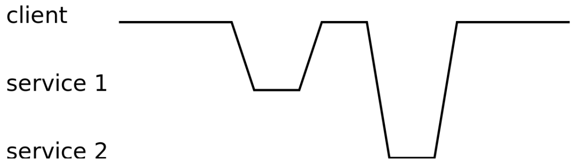
In this paper, we define a network service as a process running on a host and serving requests destined to a network socket (triple of IP address, port number, and protocol). We define dependency among services as follows. A service S_2 depends on service S_1 if a delay, degradation, or failure in service S_1 leads to a failure, disruption, or degradation of the service of S_2, directly or indirectly. Services can have different types of dependencies between each other. Chen et al. [6] classified network service dependencies into two classes: local-remote and remote-remote dependencies. Service S_1 has a local-remote dependency on service S_2 if S_1, to serve its clients, needs to contact S_2. Service S_2 has a remote-remote dependency on service S_1 if a remote client, to access service S_2, needs to access service S_1 first.

An example of a local-remote dependency is the dependency of a webserver on a MySQL server to load its con-

Figure 2: Local-Remote vs. Remote-Remote Dependency



(a) Local-Remote Dependency



(b) Remote-Remote Dependency

tents. That is, the webserver, when contacted by the client, must contact the MySQL server to load the required contents.

An example of a remote-remote dependency (indirect dependency) is the dependency between a webserver and a DNS server. A client often needs to contact a DNS server to find the IP address of the desired web server. Without the DNS server, the client most likely will not be able to contact the webserver (unless the client knows the IP address of the webserver *a priori*).

Figure 2 shows examples of remote-remote and local-remote dependencies. The graphs in this figure show the execution order of a request. The X-axis is the time axis. The graph shows the order of execution and request-response. A horizontal line shows local execution on one machine, while a downward line from one machine to the other means the first machine sent a request to the second one and is waiting for the response, and an upward line from a machine to another shows that a response has been returned.

Figure 2(a) depicts a local-remote dependency between service 1 and service 2. In this figure, the client connects to service S_1. Service S_1 in turn connects to service S_2. When service S_2 replies to the request from S_1, S_1 computes and returns the appropriate response to the request from the client.

Figure 2(b) shows a remote-remote dependency. In this figure, the client connects to server S_1 to request a service. The response from S_1 enables the client to connect to S_2.

Certainly, one can imagine more complicated types of dependencies, but they can be generalized by considering the fact that dependencies are transitive relationships. In other words, if S_3 depends on S_2, and S_2 depends on S_1, S_3 also indirectly depends on S_1.

To express the transitivity property of the dependency relationship, we define the following: $S_1 \rightarrow S_2$ means that service S_1 depends on service S_2 (either by a local-remote or a remote-remote dependency). $S_1 \xrightarrow{L} S_2$ means that service S_1 depends on service S_2, and this dependency is a local-remote dependency. Similarly, $S_1 \xrightarrow{R} S_2$ means that service S_1 depends on service S_2 and this dependency is a remote-remote dependency. Using this formalization, the trasitivity of the dependency relation can be expressed as:

$$\begin{cases} \left(S_1 \xrightarrow{L} S_2\right) \wedge (S_2 \rightarrow S_3) \Rightarrow S_1 \xrightarrow{L} S_3 \\ \left(S_1 \xrightarrow{R} S_2\right) \wedge (S_2 \rightarrow S_3) \Rightarrow S_1 \xrightarrow{R} S_3 \end{cases}$$

This property can lead to non-trivial dependencies among services that one would not suspect may depend on each other.

It should be noted that a service S_1 can have, at the same time, both local-remote and remote-remote dependencies on service S_2. For example, a webserver that is acting as a web proxy can have a remote-remote dependency on a DNS server, because the clients need to contact the DNS server to acquire the webserver's IP address before contacting the webserver. Moreover, this webserver also has a local-remote dependency on the DNS server, because it needs to contact the DNS server to acquire the IP address of the website requested by the proxy user.

In this paper, we detect both direct and indirect dependencies using a watermarking approach.

3. WATERMARKING FOR DEPENDENCY DETECTION

Dependency relationships have interesting properties that can be leveraged for detecting them. One such property is that if two services depend on each other, a delay injected in one service will propagate to the other one. We use this property for detecting dependency relationships among services.

We use a traffic watermarking approach for detecting dependency relations. That is, we perturb the connections to one service (create a ripple) and then observe whether the perturbations are propagated to a second service. More precisely, to find whether two services depend on each other, we delay the first packet of each connection directed to a service S_1. Then, we check if the connections to the other service exhibit any similar delays. Our approach is generic and application-independent, as we do not inspect or change the packet contents.

In the following section, we describe how perturbations propagate among interdependent services.

3.1 Watermarking Effect on Different Types of Dependencies

When two services depend on each other, we expect that a delay in connection to one service will result in a similar delay connection to the other service. However, the way in which the delay propagates and its direction depend on the type of the dependency. Interestingly, this fact can be used to further distinguish the type of the dependency between services.

Figure 3: Effect of delays on remote-remote dependency

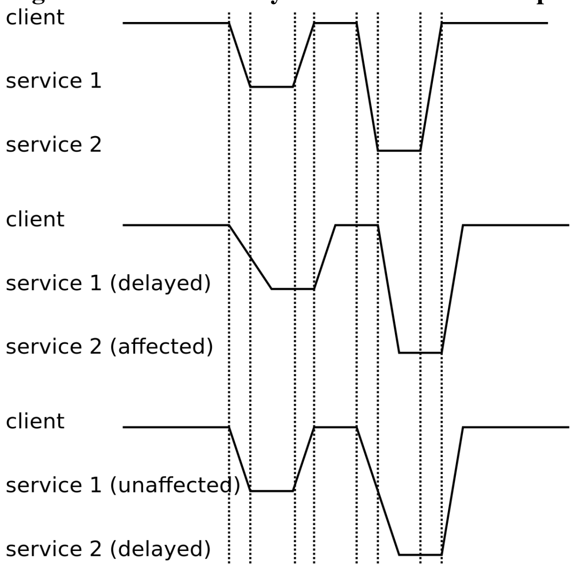
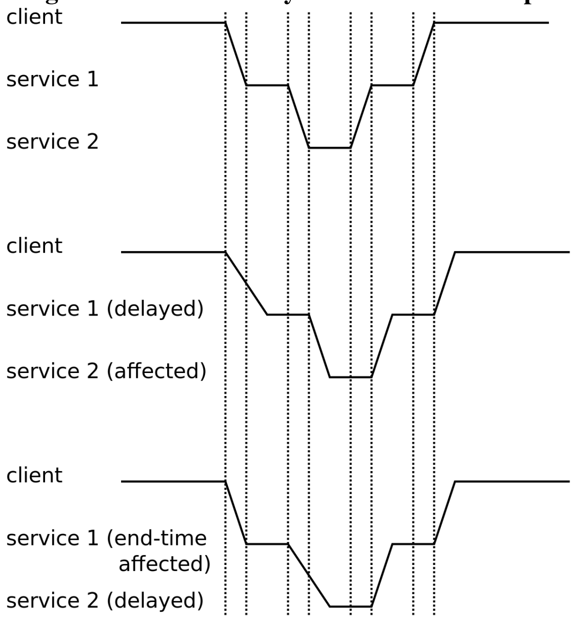


Figure 4: Effect of delays on local-remote dependency



As mentioned earlier, service S_2 has a remote-remote dependency on service S_1[1] if clients need to contact service S_1 before contacting service S_2. A classic example is a client that needs to contact a DNS service before contacting a webserver. In this case, a delay in the beginning of the connection to the DNS service (S_1) results in a similar delay of the start time of a connection to the webserver (S_2). On the other hand, a delay in the beginning of a connection to the webserver (S_2) does not have any effect on the connections to the DNS service (S_1). Figure 3 illustrates this concept.

Service S_1 has a local-remote dependency on the service S_2 if service S_1 contacts service S_2 whenever it is contacted by a client. An example of a local-remote dependency is a webserver that needs to contact a MySQL server to load the contents for satisfying a client request. In this case, a delay of the beginning of a connection to S_1 results in a similar delay of the start time of a connection to S_2. In addition, a delay of the start of a connection to S_2 results in a similar delay of the end time of a connection to S_1. Figure 4 shows this type of dependency.

To leverage these properties to detect dependencies, we need to detect the resulting delay in the depending service. In the next section, we provide the required modeling and statistical framework for detecting the propagated delay.

4. INDUCED PERTURBATION MODEL

Although the injected delay to a single connection to service S_1 leads to a similar delay in a connection to a depending or depended service S_2, recognizing the relation between these connections is not straightforward. Even though the delayed connections can be observed, they are usually mixed with many other connections, and it can be challenging to map individual connections to each other (which may require a priori knowledge about the dependency relation). Our solution is not to map individual connections, but to create patterns that statistically stand out and can be distinguished from random variations.

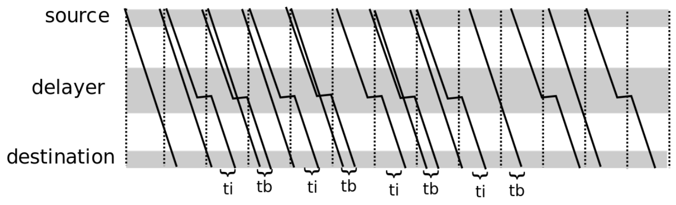
The dependency detection problem can be modeled as follows: Assume that we want to determine whether two services S_1 and S_2 have a dependency between them.[2] The goal of the analysis is to determine if any of the two services depend on the other one. Additionally, we want to determine the type and direction of the dependency.

Service S_2 has a remote-remote dependency on service S_1 or service S_1 has a local-remote dependency on service S_2 ($S_1 \xrightarrow{L} S_2 \vee S_2 \xrightarrow{R} S_1$) when a delay, degradation, or failure in service S_1 causes a delay, degradation, or failure in service S_2. Our hypothesis is that if service S_2 has a remote-remote dependency on service S_1 or the service S_1 has a local-remote dependency on the service S_2 ($S_1 \xrightarrow{L} S_2 \vee S_2 \xrightarrow{R} S_1$), a delay d_1 in service S_1 should result in a similar delay $d_2 \approxeq d_1$ in service S_2. As a local-remote dependency example, if a webserver depends on a MySQL server to deliver its contents, a delay to the connections to the webserver causes a similar shift in the arrival times of the requests to the MySQL server. In the remote-remote dependency example, if a webserver "depends" on the DNS server to be contacted by clients, a delay to the connections to the DNS server causes a similar shift in the arrival times of (a fraction of, due to caching) the requests to the webserver.

[1]Throughout the paper, the service that has to be contacted first will be called S_1

[2]We may have chosen these two services based on their high correlated activity, some heuristics, or we could have simply tried all possible pairs of services.

Figure 5: Delayer effect on connections



As we also showed in Section 3.1, this delay propagation holds regardless of the dependency type.

4.1 Detection of the Injected Delay

To detect the dependency relations, we need to detect artificially-injected delays. Assume the following scenario: A connection C_1 to service S_1 is delayed. As a result, service S_1 will contact service S_2 through connection C_2 with a delay (For the sake of simplicity, we assume a local-remote dependency. A similar argument can be used when a remote-remote dependency exists between the services). The observer will see these two connections, along with thousands of other connections, and may not be able to recognize the two connections C_1 and C_2 as being causally related.

To make the perturbation visible to the observer, we create different patterns in the request arrival times at a selected service. These patterns are different in different time windows, but result in similar patterns in the related service.

To model the service activity, we divide the observation period into time windows of equal size ($w_1, w_2, \ldots, w_{2n}$, where $|w_i| = s_w$, in which s_w is the window size, and $2n$ is the number of time windows). We delay the requests directed to service S_1 for t_d in odd time windows $w_1, w_3, w_5, \ldots, w_{2n-1}$ and we do not delay them in even time windows. This process will create time windows with more than average requests (t_b busy time window) and time windows with less than average requests (t_i idle time window) on S_2 (also referred to as *ripples*), as shown in Figure 5. It is straightforward to show that the size of idle time windows and busy time windows is equal to the amount of delay ($t_i = t_b = t_d$). The smaller the time window size, the more samples can be gathered during a fixed period of time. On the other hand, the time window size, s_w, should be large enough to separate the effects of consecutive tests and make the samples independent of each other. In other words, the delayed packets (and the connections triggered by them) should have enough time to reach their destination before the next period of delaying starts.

Let's assume that the number of requests for service S_2 in different time windows (t_d) follows an unknown distribution $D_0 = D(\mu_0, \sigma_0)$, with the mean and standard deviation equal to μ_0 and σ_0, respectively. Also, assume that ρ is the fraction of requests destined to S_2 that are caused by requests destined to S_1.

When requests destined to S_1 are delayed in the described way, the number of requests in the idle time windows and busy time windows on S_2 follow
$D_1 = D(\mu_0 \cdot (1-\rho), \sigma_0 \cdot (1-\rho))$ and
$D_2 = D(\mu_0 \cdot (1+\rho), \sigma_0 \cdot (1+\rho))$[3], respectively. In other words, this watermarking results in consecutive periods of length t_d of distributions D_1 and D_2 separated from each other by periods of length $w - t_d$. The request arrival distribution for the time period between the busy and idle time periods follows distribution D_0. In the next section, we show how these artificially-generated patterns can be distinguished from random noise with high levels of confidence.

5. STATISTICAL INFERENCE

To show the dependency of two services, we want to reject the hypothesis that two services are independent. Therefore, we use statistical hypothesis testing for showing the existence of the dependency relationship. First, we assume that the given services are independent. In other words, injecting delays in one service does not alter the request arrival time distribution on the second one (null hypothesis). Then, we compute the conditional probability of the observed request arrival time samples, given the null hypothesis. If the probability of the observed sample, given the null hypothesis, is lower than a threshold, the null hypothesis is rejected and the dependency between services is assumed.

While we use the described request delaying scheme, we use different statistical tests for comparing the means of the two populations and to reject the null hypothesis, including: *two independent samples means t-test*, *two dependent samples means (paired) t-test*, and *two dependent samples (Wilcoxon) signed rank test*.

To simplify the formalization, and without loss of generality, we assume that we want to determine the relationship between services S_1 and S_2, while we are delaying requests destined to S_1. To describe the experiment, we use the following variable definitions:
X is the random variable for the number of requests arriving to service S_2 in each time window of length t_d, when no delay is applied[4].
X^i and X^b are the random variables for the number of requests arriving for service S_2 in each t_i and t_b time windows, respectively.
μ_i and μ_b are the mean of X^i and X^b, respectively.
Our null hypothesis is that the busy and idle time windows have the same average request arrival rates: $H_0 \equiv \mu_i = \mu_b$. The null hypothesis states that S_2 is independent of S_1, and, as a result, injecting delays to requests to S_1 does not change the request arrival distribution in S_2 ($H_0 \Rightarrow \mu_i = \mu_b$ or equivalently $\mu_i \neq \mu_b \Rightarrow H_1$).

[3]If $X \sim D(\mu_x, \sigma_x)$ and $Y = aX$ then $\mu_y = a \cdot \mu_x$ and $\sigma_y = \sqrt{\frac{\sum (y_i - \mu_y)^2}{n}} = \sqrt{\frac{\sum (a \cdot x_i - a \cdot \mu_x)^2}{n}} = a \cdot \sigma_x$
[4]We use sliding time windows

We describe several statistical tests to calculate $Pr\left(e|\mu_i = \mu_b\right)$, in which e is an observed test statistic. In the rest of this section, we describe each statistical test and its properties.

5.1 Two Independent Samples Means t-Test

This test computes the probability that the distributions from which two samples are drawn have the same means.

The p-value of the 2-sample t-test is calculated using the following formula:

$$t = \frac{\left(\overline{X^b} - \overline{X^i}\right) - (\mu_b - \mu_i)}{\sqrt{\frac{S_p^2}{n_b} + \frac{S_p^2}{n_i}}}$$

$H_0 \Rightarrow \mu_i - \mu_b = 0 \Rightarrow t = \frac{\overline{X^b} - \overline{X^i}}{\sqrt{\frac{S_p^2}{n_b} + \frac{S_p^2}{n_i}}}$ follows t-distribution with $df = n_b + n_i - 2$ degrees of freedom, where, $S_p^2 = \frac{(n_b-1)\cdot S_b^2 + (n_i-1)\cdot S_i^2}{n_b+n_i-2}$ and $S_i^2 = \frac{SS_i}{n_i-1}$ and $S_b^2 = \frac{SS_b}{n_b-1}$.

We already showed that if $S_1 \xrightarrow{L} S_2 \vee S_2 \xrightarrow{R} S_1$, $\mu_i = \mu_0 \cdot (1-\rho)$ and $\mu_b = \mu_0 \cdot (1+\rho)$. Using the *central limit theorem*, it can be shown that, regardless of how small ρ is, an arbitrarily small p-value can be obtained given a large-enough set of samples. In other words, the following holds:

$$H_1 \Rightarrow \mu_b \neq \mu_i \Rightarrow \begin{cases} \overline{X^b} - \overline{X^i} \xrightarrow[n\to\infty]{} \mu_b - \mu_i \\ \sqrt{\frac{S_p^2}{n_b} + \frac{S_p^2}{n_i}} \xrightarrow[n\to\infty]{} 0 \end{cases} \Rightarrow$$

$$\Rightarrow t = \frac{\overline{X^b} - \overline{X^i}}{\sqrt{\frac{S_p^2}{n_b} + \frac{S_p^2}{n_i}}} \xrightarrow[n\to\infty]{} \pm\infty \Rightarrow p - value \xrightarrow[n\to\infty]{} 0.$$

On the other hand, if the two services are independent, regardless of the size of the sample, we will not see small p-values. To summarize, this test is reliable as long as a large-enough sample set is available. However, this test is sensitive to noise, because small number of data points with extreme values can skew considerably the test results.

5.2 Two Dependent Samples Means Paired t-Test

The number of requests that arrive at S_2 in each idle time window (x^i) is related or dependent on its neighbor (consecutive) busy time window (x^b), as servers have different load/request arrival behavior during different times of the day. Unfortunately, the *two independent samples means* test does not take advantage of this property.

If we consider each consecutive value of t_i and t_b to be related samples, we can use the paired sample t-test. If we define $D = X^b - X^i$, the following variable, t ratio, follows Student's t distribution with $df = n - 1$[5] :

$$t = \frac{\overline{D}}{\frac{S_D}{\sqrt{n}}} = \frac{\overline{D}}{\sqrt{\frac{SS_D}{n(n-1)}}} \text{ [8]}$$

It should be noted that if the null hypothesis is not true, increasing the size of the sample set increases the power of the test.

[5] *df*: degree of freedom

$$H_1 \Rightarrow \mu_b \neq \mu_i \Rightarrow \begin{cases} \overline{D} \xrightarrow[n\to\infty]{} \mu_b - \mu_i \\ \sqrt{\frac{SS_D}{n(n-1)}} \xrightarrow[n\to\infty]{} 0 \end{cases} \Rightarrow$$

$$\Rightarrow t = \frac{\overline{D}}{\sqrt{\frac{SS_D}{n(n-1)}}} \xrightarrow[n\to\infty]{} \pm\infty \Rightarrow p - value \xrightarrow[n\to\infty]{} 0$$

Therefore, the power of this test increases as the number of the samples increases. This property is important, because if $\mu_b - \mu_i$ is relatively small, the difference between the distributions can still be distinguished by increasing the size of the sample set (i.e., by running the experiment for a longer time).

This test is less sensitive to noise (extremely high or low values in the samples) than the *two independent samples means test*, because extreme sample data values increase or decrease both the nominator and the denominator of the fraction in t formula. But, a small number of extreme values still can skew the test results considerably, because the absolute value of the sample data points are used in computing t.

5.3 Two Dependent Samples Means (Paired Wilcoxon) Signed Rank Test

One would expect the network behavior to change through time. For example, a university web server may be busier in specific times of the year (e.g., during the registration period). Nevertheless, one would also expect that the network behavior of a service would be rather similar in two close time periods. The two independent sample t-test does not take advantage of the fact that t_i (idle time windows) and t_b (busy time windows) samples are pairwise related. That is, the two independent sample t-test ignores the order of the sample data points. In contrast, the *two dependent samples means t-test* uses this ordering information, but it remains sensitive to noise, because it uses the absolute value of the sample data points in computing t.

An alternative statistical test that takes pairwise dependency between samples into account is the Wilcoxon test. The Wilcoxon test checks whether or not paired samples of t_i and t_b are drawn from the same population. In this approach, we match each t_i to its consecutive t_b. In the null hypothesis, we consider t_is and t_bs as samples of the same population. In other words, if service S_2 does not depend on S_1, delaying requests to service S_1 should not create any changes in the distribution of the requests to S_2.

To prove that service S_2 depends on service S_1, it is sufficient to show that the number of requests received on service S_2 at t_is does not follow the same distribution as the number of requests received at t_bs. Because the t_is and t_bs are paired and related, we use Wilcoxon signed-rank test to calculate the z-score for the null hypothesis (that t_is and t_bs belong to the same distribution).

In this test, each X^i is paired with an X^b value. The k^{th} pair is denoted as X_k^i, X_k^b.
$D_k = |X_k^i - X_k^b|$, $W = \left|\sum_{k=1}^{n} sgn\left(D_k\right) \cdot R_k\right|$, where R_k is the rank of D_k when the list is sorted in ascending order. A z-score can be computed using $z = \frac{W - 0.5}{\sigma_W}$, where

$$\sigma_W = \sqrt{\frac{n(n+1)(2n+1)}{6}}$$

It can be shown that if the two distributions are different ($\mu_i \neq \mu_b$), D has a distribution with a non-zero average.

Please notice that $H_1 \Rightarrow \mu_i \neq \mu_b \Rightarrow z \xrightarrow[n\to\infty]{} \infty \Rightarrow$
$\Rightarrow p - value \xrightarrow[n\to\infty]{} 0.$
In other words, regardless of how small the fraction of the delayed requests (ρ) is, arbitrarily small p-values can be achieved by increasing the length of the experiment (period of time).

Finally, we report service S_2 to depend on service S_1 if any of the three statistical tests can reject the null hypothesis.

5.4 Environment Effects on the Accuracy of the Statistical Tests

There are several factors in a real network environment that can limit the accuracy and power of the proposed statistical tests: low number of requests to the server; low percentage of the requests to the server affected by the delayer; and jitter in the network.

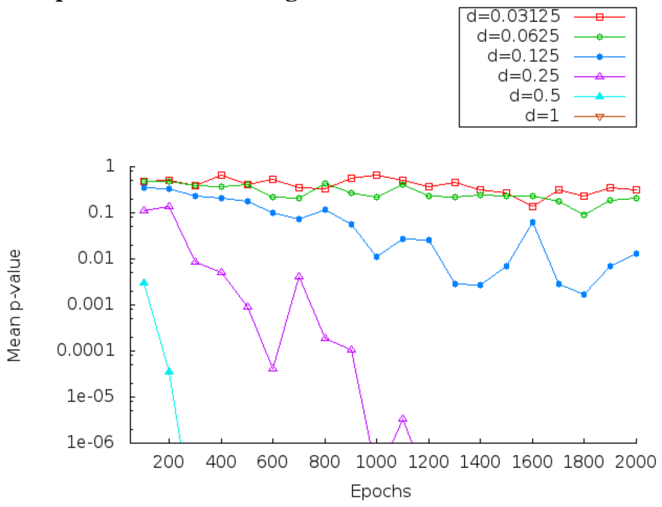
Low number of requests to the server results in too few slots with non-zero data points, preventing the statistical tests from differentiating between two distributions. This problem can be mitigated by processing data over a longer period of time. In other words, if a sample of size $l \cdot n$ of X has a small mean of μ, it can be converted to a sample of size n of a distribution X' with mean $\mu' = \mu \cdot l$, simply by the following variable: $X'_i = \sum_{j=l\cdot i}^{l\cdot i+l-1} X_j$.
Low percentage of the requests to the server affected by the delayer blurs the difference between the two distributions (with and without delay). In all three proposed statistical tests, we showed that this problem can be resolved by increasing the length of the experiment.
Jitters in the network cause the requests that were supposed to arrive in the same time period to arrive in different time periods and vice versa. If we assume that the packets that would arrive in a wrong time slot because of the jitter in the network are selected randomly based on some probability, this problem can be converted to and modeled as the previous problem (Low percentage of the requests affected by the delayer). Therefore, increasing the length of the experiment and hence, the sample set size, mitigates this problem.

The limits and powers of the statistical tests can be measured using simulation. To measure the accuracy of the proposed statistical tests, we simulate servers with different properties, such as different request arrival time distributions and different percentage of clients under delay.

6. TEST ACCURACY ANALYSIS

To compute the accuracy of the tests, in general type I and type II errors are presented. The type II error is equivalent to a false positive error and its value is directly computed as the p-value. The type I error is equivalent to a false negative error: t is more challenging to calculate, and it is not always possible to derive a formula for it analytically. For example, to the best of our knowledge, no analytical solution to Wilcoxon paired test exists. Therefore, we calculate the test power by simulation.

Figure 6: The effect of delay percentage and the length of experiment on false negatives



We calculated the probability of rejection of the null hypothesis (detection) under different assumptions about the distributions of the arriving requests. One main finding is that the power of the experiment depends on different parameters such as the number of samples, difference of the distribution means, difference of distribution standard deviations, and percentage of the requests being delayed.

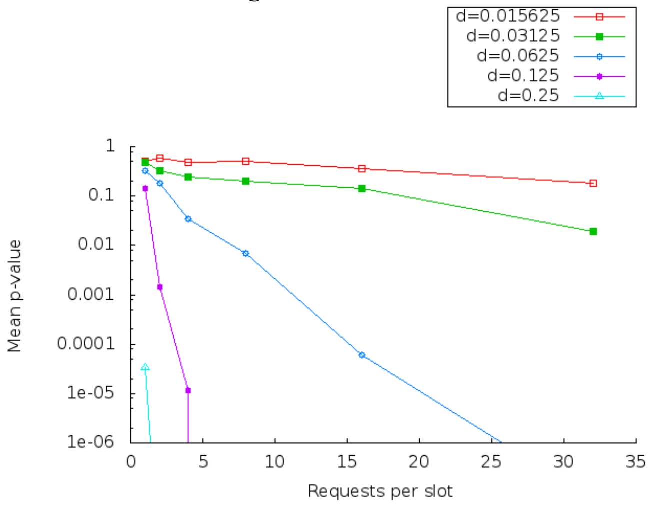
We calculated the power of the tests both by completely simulating the server and by simulating delays in real server traffic. We use a Poisson distribution for the arrival times of the simulated requests.

6.1 Simulation

To study the effects of different parameters (characteristics of the network under analysis) on the accuracy of the tests, we ran a set of simulations. First, we simulated the request arrival times for a set of services based on a Poisson distribution. We varied the percentage of requests getting delayed by the delayer, the load of the server, and the length of the experiment (amount of sample data).

Figure 6 shows the p-values of the true dependencies. The X-axis corresponds to the length of the experiment (number of periods in which the service was delayed). The Y-axis shows the p-value of the true dependency calculated by the statistical tests in log scale. Each line corresponds to a different request delay percentage (the percentage of the requests that are affected by the delayer). We used simulated service with an average load of 1 request per time slot, and varied the length of the experiment and the percentage of delayed requests. As Figure 6 shows, the results improve when more

Figure 7: The effect of delay percentage and the load of the service on false negatives



sample data points are added or when the percentage of delayed requests increases.

Figure 7, similarly, shows the effect of the load of the server on the test power. The X-axis corresponds to the load of the service (in requests per slot). The Y-axis is the p-value of the true dependencies in log scale. The experiment contains 200 data points (epochs). This figure shows that the p-value decreases as the delay percentage or the service load increases.

In addition to this completely simulated experiment, we used real network NetFlow records and simulated the delay injection on real server request arrival times. Similar results were achieved with the simulated services.

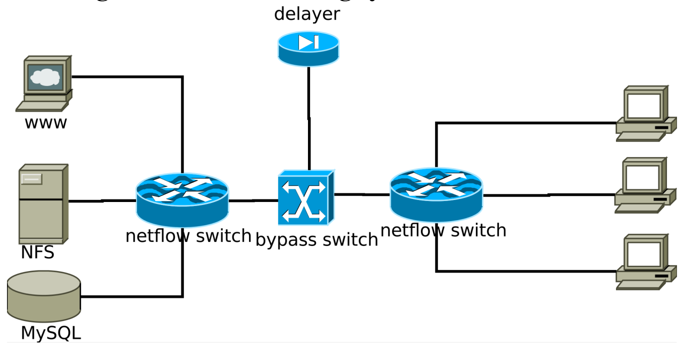
6.2 Sanity Check

To test the accuracy of the test, we also ran the statistical tests on the data while we were not delaying any packets in the network. We used NetFlow data gathered during one month. That is, since no packet has been delayed by our delayer in this data, we expect the request arrival of consecutive time slots to be drawn from the same distribution. Consistent with our prediction, no dependency among services was detected on this dataset by any of the statistical tests.

7. SYSTEM ARCHITECTURE

Figure 8 shows the architecture of the watermarking system. As shown in the figure, our dependency detection system is composed of two main components: the delay injection device and the network data gathering device (shown as NetFlow switches in the figure). Given the fact that the delayer is built using commodity hardware, we added a bypass switch to increase the reliability of our final solution. A bypass switch is a special network switch with at least four network interfaces. It intercepts the connections on each of

Figure 8: Watermarking system architecture



its input interfaces and forwards them to the delayer. The delayer also acts as a switch. The delayer can decide when to forward the packets (whether or not to delay any specific packet). When the delayer forwards the packet to its other interface, the packet is received by the bypass switch and is forwarded to the appropriate network interface. The bypass switch checks the availability and responsiveness of the delayer using a heartbeat protocol. In case the delayer becomes unresponsive, the bypass switch bypasses the delayer and becomes a simple switch connecting its two interfaces directly.

7.1 Delayer

The delayer should be located on the path between services under study and their clients. It selectively delays packets sent to those services. By selectively delaying or not delaying packets, the delayer creates statistically detectable request arrival patterns at the servers under examination.

Naturally, the more traffic that passes through the delayer, the more power it has for creating request arrival patterns on more services. The delayer is ideally a switch that resides in the path of the dependencies one wants to detect. For example, the core switch of an organization would be a good candidate for detecting the dependencies among services in different buildings, while a switch close to a server room would be a good candidate for detecting the dependencies to the services in that room.

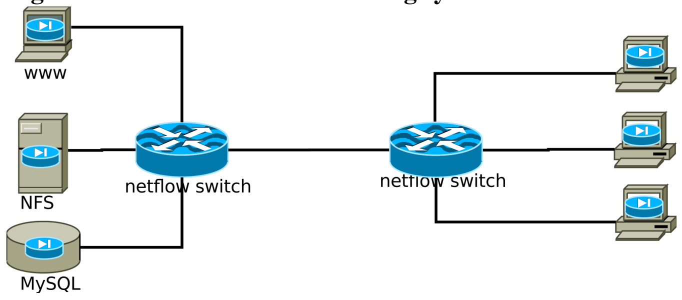
7.2 Network Data Collector

The network data collector is used for collecting the following information about each connection to or from the services under study: the service socket address (the tuple of IP, port, and protocol), the connection start time, and the connection end time.

We purposely chose to impose minimal requirements on the network data collector. The reason is that complete information about the network data is not always available or achievable, due to network privacy policies or technical challenges.

First, storing complete network data (including the payload) requires a significant amount of storage. Therefore,

Figure 9: Distributed watermarking system architecture



network administrators usually adopt less complete formats of network data dumps that capture important and compact information about the traffic. NetFlow is one example of this summarized network data format. NetFlow, for each connection, keeps only the socket address of both sides, the number of bytes and packets sent and received, and the timing of the connection start and end. Dumping only packet headers using tcpdump is another example of summarized network datasets.

Second, even if full packet data is available from a network, the network privacy issues and policies may prohibit a researcher or data analyst from having access to this data.

Therefore, to be widely applicable, a traffic analysis tool must rely on limited amounts of information.

7.3 Handling Lack of Infrastructure for Central Analysis

In some cases, installing a central delayer or network data collector can be impossible or challenging, due to the policy or technical issues. In those cases, the delay injection and data gathering can be achieved using host-based delayers and sensors, as shown in Figure 9. In other words, the sensor and the delayers directly reside on the servers under analysis and gather the data on the host. Although this can lead to clock discrepancy[6] among multiple delayers and would require additional considerations to resolve that issue, it is a last resort for the cases where network-based solutions are not deployable.

8. IMPLEMENTATION

We implemented and tested a central prototype (using a central network-based delayer) of *Rippler* in a small lab. We introduce the minimal amount of delay that is required to detect perturbations. This delay time should be greater than the clock discrepancy between the delayer and flow collector devices. The clock discrepancy in our network is less than or equal to 40 milliseconds (the computer clocks are synchronized by NTP). We used a delay of 100 milliseconds, which one can expect not to have significant effects on typical services. We are aware that there may exist services for which 100 milliseconds of delay could cause a failure, but these services are usually not implemented in typical TCP/IP networks. These services should have their own dedicated networks as small amounts of delay/jitter are expected in regular networks.

In our prototype implementation, we were able to show that the busy and idle time windows (t_bs and t_is) are detectable.

We developed the delayer using an off-the-shelf Linux system. This system has two bridged interfaces and acts as a regular bridge for the packets that are not selected to be delayed. On the other hand, the packets that are selected to be delayed are buffered and forwarded at a later time.

A set of five services were selected for delay injection. We ordered the set of selected services and periodically delayed them one after another. The first packets to each service, when it is its turn, is delayed for 100 milliseconds in a period of 10 seconds and then no service is delayed for 10 seconds and the next service is delayed afterwards. The periods of no delay between the delay periods of different services is to separate (and make recognizable) the effects of injected delays in each service from others.

8.1 Installation and Detected Dependencies

In the prototype application, we showed that the injected delays are recognizable. For the main experiment, we installed our delayer and NetFlow collectors in a university department lab. Unfortunately, the university network administrators were not able to provide us with a central delay injection point in their infrastructure. Therefore, we were forced to deploy our host-based delayer. The delayer was installed in a lab used by students (mainly for doing their assignments and homeworks). To overcome the clock discrepancy between the hosts that run the delayer, we had to increase the amount of delay from 100ms to 500ms (the delay discrepancy between the hosts was around 40ms). A set of 54 most frequently used services were selected to be delayed.

In a normal day, students use the computers in this lab to do their homework. All the machines in the lab are centrally managed (using cfengine) and they have an identical configuration. Users authenticate using an LDAP server, and their home directories are mounted from several NFS servers. Users check and send emails using an internal mail server. An internal DNS server is used to look up IP addresses. All hosts have an /etc/hosts file that lists all the internal servers' names along with their IP addresses.

Network administrators provided us with a central network data gathering point. The network traffic information is gathered in NetFlow format. We gathered 133GB of NetFlow data which corresponds to 12.5 billion connections. The packets were delayed for 500ms.

[6]Using host based delay injection leads to a clock discrepancy problem. That is, different delayers on different machines will start and stop delaying packets on slightly different times, due to the clock skew of their respective machines.

Table 1: The dependency analysis results

service	perturbated services
NFS1	LDAP, web, cfenginei, dhcp, portmapper, lab shell, DNS
web26	NFS26, portmapper26
NFS46	LDAP12, LDAP36, NFS13, dhcp10, NFS41
CFengine	NFS1, NetBios, IMAP, NFS2
NFS3	MySQL and NFS4
WWW	NFS1, IMAP, NFS4

In the course of the experiments, we delayed requests to 54 services. We compared our results with three previous approaches. We first present the comparison of our approach to these approaches in Section 9. Later in Section 10, we describe previous work in dependency detection and compare the characteristics of our approach against the previous work. Table 1 shows the detected true dependencies using *Rippler*.

9. COMPARISON WITH SHERLOCK, ORION, AND NSDMINER

We ran Sherlock [3], Orion [6], and NSDMiner [13] (three passive dependency detection systems) on our NetFlow dataset gathered from the department computer lab. The results of the experiment are shown in Figure 10. We ran all four systems with different parameter tunings and calculated the number of false positive and true positives for each tool, and for each configuration. As shown in Figure 10, *Rippler* produces less false positives for any given true positives that it generated. It should be noted that *Rippler* did not generate any false positives when we set the p-value to any value less than or equal to 10^{-6}.

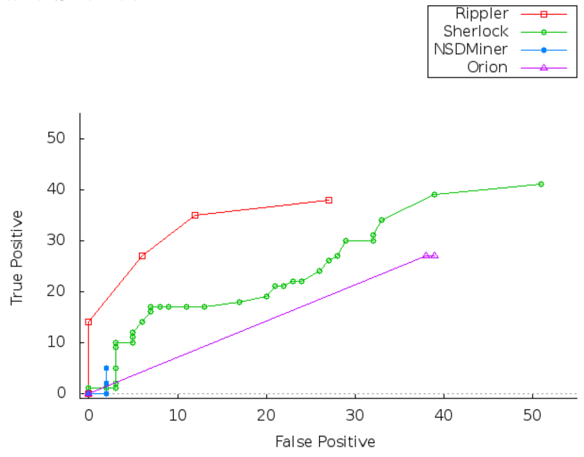
To verify and compare the results, we manually labeled 156 dependencies (the superset of all resulting dependencies from all four tools). The dependencies were confirmed by interviewing the administrators. We also looked into the configuration of the hosts in the lab. As all the hosts share the same configuration, by looking into the host configuration we learned about many dependencies.

In the end, 68 of these 156 dependencies were true dependencies, 70 were false dependencies, and we were not able to determine the correctness of 18 dependencies.

9.1 Sherlock

Sherlock calculates the strength of a dependency relation from service S_2 to service S_1 as the probability that service S_1 accessed within a time interval from when service S_2 is accessed. Among other problems, this approach will detect every pair of frequent services as depending on each other. The results of our experiments verified this property. Sherlock created a large number of false positives, and typically these false positives included the most-frequently-used services.

Figure 10: ROC curves for Rippler, NSDMiner, Orion, and Sherlock



It should be noted that we recognize the fact that we tested a partial implementation of Sherlock, using only the part that detects the service dependencies. Sherlock, in addition, uses this information to predict system failures and localize faults.

9.2 Orion

Orion exploits the fact that if two services are depending on each other, the delays between consecutive accesses follow some pattern. For example, if an application needs to access service S_1 before accessing service S_2, the delay between accesses to service S_1 and service S_2 will follow some non-random distribution. Orion uses this property to detect this different distribution from a random distribution. Although Orion has a more compelling confidence measure than Sherlock, it still fails to create high-confidence dependencies. Orion confidence in a dependency relationship is expressed in how different the delay patterns between accesses to the two services are from random, in terms of number of standard deviations. In our experiments, Orion did not generate any dependencies for any confidence higher than 1.5 standard deviations, which corresponds to $p - value = 0.1336$.

9.3 NSDMiner

NSDMiner detects only local-remote dependencies. Therefore, it misses remote-remote dependencies. Another problem with NSDMiner is the fact that it is sensitive to the timing information of the sensors. NSDMiner detects a dependency from service A to service B when the probability that the life span of connections to service B is included in the life span of a connection to service A is higher than a threshold. This threshold is called α. We varied α from 0 to 1.

In our experiment, the NetFlow probes and the delayer are placed between the lab clients and the servers they use, which are located in a server room. Therefore, most of the traffic from a server to another does not pass our probes and delayers. This property makes the remote-remote dependencies the most common dependencies in our configuration. NSDMiner does not try to detect remote-remote dependencies, and therefore it misses most of the true dependencies in our configuration.

9.4 Correlation Does Not Imply Causation

The most common problem with correlation-based approaches for dependency detection is the confusion of correlation with causation. In other words, passive approaches are susceptible to false positives. The experiments showed that the problem exists in the approaches we tested.

For example, the correlation-based approaches detected a dependency from several services to the main DNS server. This dependency is a false positive, because the machines that are used in the experiment have the IP addresses of the hosts in the internal network in their /etc/hosts file. Therefore, these hosts do not need to lookup the IP addresses of the internal services. The reason for the false positive is that DNS is one of the most frequently used services in the network, and therefore it appears as the prequel to other services being used. This property of the DNS server has led to false dependencies detected by both Orion and Sherlock. NSDMiner does not detect dependencies to the DNS server, because NSDMiner only detects local-remote dependencies[7].

9.5 *Rippler*

In the experiment, we excluded the services that did not receive at least 1000 requests in the period of data gathering. We varied p-value from 1 to 10^{-20}. We experienced some false positives when p-value was greater than 10^{-6}, but when we reached 10^{-6}, all false positives disappeared.

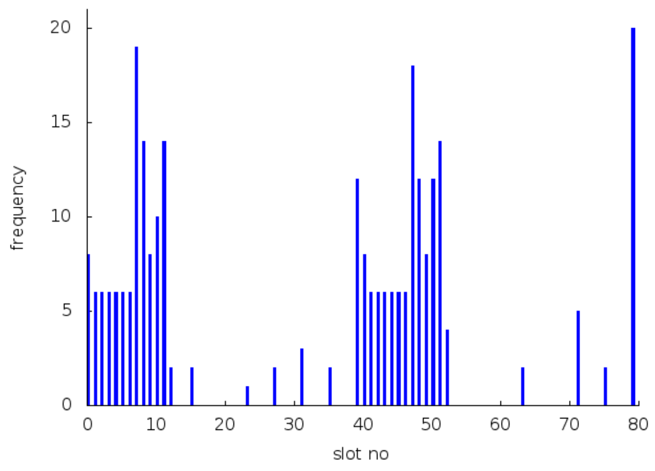
When we included the idle services (services with less than 1000 requests during the experiment), seven false positives appeared.

We did not expect any false positives from *Rippler*. The reason for this confidence is that when a ripple is detected the probability of it being caused by a random process is negligible (by choosing a small enough p-value as the threshold). One possible explanation is that the dependency between two services has a long time lag and therefore the delay ripple from the first service arrives late and is considered as the effect of another service.

Figure 11 shows the distribution of the detected ripples among the time slots. Each window of time dedicated to a service is called a period. Each time slice inside a period that is used to detect a ripple is called a time slot. As a period is 20 seconds long and we use 500ms of delay, we have 80 slots in each period (because of using sliding windows). As shown in the figure, the ripples are either in the left side of the period or in the middle of it. Even though most ripples follow this pattern, which is expected from a low-latency dependency, we observe several cases of ripples scattered in

[7]the dependency between the services and DNS server is usually a remote-remote dependency

Figure 11: Distribution of detected ripples in different time slots



random slots. This shows that high-delay dependencies exist among our services. These high-delay dependencies can be the cause of false positives, because they confuse *Rippler* in recognizing which service delay was responsible for the observed delay in the target server. This problem can be easily resolved by dedicating a longer time period to each service.

10. RELATED WORK

Previous work on service dependency detection includes many different approaches. In order to compare the previous work, we first introduce some desirable effectiveness criteria for a dependency detection approach.

10.1 Dependency Detection Effectiveness Metrics

One can expect several properties from an ideal dependency detection system. The dependency detection should: be able to detect direct and indirect dependencies; be able to handle partial data; affect the network operation minimally; be easy to deploy; be application-independent; require a minimum amount of change to machines in the network; provide a metric of meaningful confidence; need a minimum amount of high level data; and be able to work with anonymized data.

A dependency-detection approach may or may not detect both local-remote and remote-remote dependencies.

A dependency detection approach may or may not work with partial data. Partial data issues occur when the whole information flow path from the depending service to the depended service is not visible to the system. For example, assume that service S_4 depends on service S_3, which in turn depends on service S_2, which finally depends on service S_1. If the communication between service S_2 and S_3 is hidden from the system, the dependency detection system will not be able to build the entire information flow path between service S_1 and service S_4 and may miss the dependency relationship.

A dependency detection approach can be host-based or network-based, depending on the data source it uses (network traces or system logs). A host-based system sits on the host on which the service dependencies are going to be explored and has access to the host configurations and various types of logs generated by the host. On the other hand, a network-based approach looks into the traffic.

A dependency detection approach can be application-dependent or application-independent. An application-independent approach works correctly in presence of unknown types of services. On the other hand, the application-dependent approaches need readjustments or reimplementations for different types of services.

A dependency detection approach may or may not handle cached services. We call a service cached if its clients normally cache the response from the server, and use the cached response in case of an identical request at a later time. Cached services pose a challenge to dependency detection approaches as they invalidate the assumption that every request to the depending service will cause a request to the depended service.

A dependency detection approach may or may not provide a measure of confidence for detected dependencies. To have a confidence level for each dependency detection can help the administrators to decide and choose their most important dependencies based on the most reliable information.

And finally, a dependency detection approach may or may not be susceptible to confusing correlation with causation.

10.2 Previous Work

NSDMiner [13] is a passive correlation-based dependency-detection system. It looks only for local-remote dependencies by computing the probability of the remote service being requested given the local service being requested.

eXpose [11] is another passive dependency-detection system. It uses JMeasure as a metric for measuring the dependency of two services. eXpose uses statistical rule mining to detect frequent patterns of communication between services. As it detects correlation between services, it is also susceptible to false positives. eXpose needs to see the information flow path between the correlated services and therefore, cannot handle partial data problem. It also suffers from false positives, because it is a correlation based technique.

Chen et al. [6] developed a passive dependency-detection system called Orion. Orion uses traffic delay distributions to find services that depend on each other. More specifically, Orion looks for spikes in delays of service usage. The basic idea is that the delay patterns (spikes) between two independent services are expected to be random, while the delay distribution between two depending services follows some distribution that depends on the execution path of the services. As Orion is a correlation detection technique, it also suffers from the false positive problem. For example, if two services depend on a third service, and that service is causing the same spikes to propagate to both services, Orion detects those two services as depending on each other, while in reality, they depend on a common third service.

Sherlock [3] is a passive host-based dependency-detection system. Sherlock fails to detect indirect dependencies, because it recognizes dependencies between two services only when the same client (on which Sherlock processes are run) directly contacts both services. It does not detect local-remote dependencies. It also has difficulties to detect dependencies among cached services, as it computes the strength of the dependency as the probability of the services being accessed together. Sherlock also suffers false positives, because it detects correlated services as depending.

Pinpoint [5] is a host-based active dependency-detection system that uses system logs to trace the requests across a distributed system. It uses a probabilistic context-free grammar to model normal path behavior. Pinpoint modifies the service under study to generate unique IDs for each request and pass them through the system. Pinpoint is application-dependent, and therefore, it cannot be used to detect dependencies among unknown services.

Macroscope [14] is a passive host-based dependency-detection system. It uses system logs to map network connections to different applications/processes. The analyzer aggregates the information gathered on different hosts to extract the dependencies between different applications.

Constellation [2] is a passive dependency-detection system that uses activity correlation as a measure of dependency. It uses statistical hypothesis testing to calculate confidence of the derived dependency relations. It learns the delay distribution function based on two predefined distribution function classes (Gaussian and exponential). It uses expectation maximization to compute the output channel distribution based on the input channel distributions.

X-Trace [10] is an active dependency-detection system that modifies network protocols to carry X-Trace meta-data. It inserts unique identifiers into the requests and propagates them to the further requests generated by the original one. Then, X-Trace gathers this information and builds a tree structure of the request path. X-Trace enforces a tree structure for request path topology. Therefore, it cannot detect dependencies of other topologies.

Kind et al. [12] used a passive correlation-based approach to detect direct and indirect dependencies between different services in a corporate network.

Dechouniotis et al. [9] developed a passive network-based dependency-detection system. They used NetFlow as their data source, and used a fuzzy inference engine to classify the detected relations as high confidence and low confidence relations.

ADD [1, 4] (Active Dependency Discovery) uses active perturbation in an to detect dependencies between services.

Table 2: Comparison with Dependency Mining Tools
Table column names correspond to the following properties: handling Cached Services (CS), handling both Local-remote and Remote-remote dependencies (LR), handling Bad Sensor Placement (SP), No modification to the Hosts (NH), No extra Traffic injected (NT), Application Independence (AI), Not Confusing Correlation with Causation (NCC). An 'N' means that the corresponding system handles the corresponding problem, and a 'Y' means that it fails to completely handle the corresponding problem.

Approach	CS	LR	SP	NH	NT	AI	NCC
NSDminer [13]	Y	N	N	Y	Y	Y	N
eXpose [11]	Y	Y	N	Y	Y	Y	N
Orion [6]	Y	Y	N	Y	Y	Y	N
Sherlock [3]	Y	N	N	N	Y	Y	Y
Pinpoint [5]	Y	Y	N	N	Y	N	Y
Macroscope [14]	Y	N	N	N	Y	Y	N
Constellation [2]	Y	Y	N	N	Y	Y	N
X-Trace [10]	Y	Y	Y	N	N	N	Y
ADD [4]	Y	Y	Y	N	N	N	Y
Rippler	Y	Y	Y	Y	Y	Y	Y

This approach uses a relatively aggressive approach as it perturbs different components of the system by load injection. ADD creates some workload on a component in the network and observes its effect on another component. To create appropriate workload for a service, ADD needs to understand the logic of the service. Therefore, ADD is application-dependent. ADD also has problems detecting dependencies to replicated (or load balancing) components, as adding load to one component may not necessarily lead to reduced efficiency of the target service.

In summary, all previous passive approaches are susceptible to confuse correlation with causation. Therefore, all previous passive approaches have high false positive rates (compared to active approaches). However, all previous active approaches are application-dependent, cannot be used to detect dependencies among unknown services, they should be customized for different services.

Table 2 shows and compares the features of the previous work in service dependency detection. *Rippler* is, to the best of our knowledge, the first application-independent active dependency-detection system.

We selected Sherlock, Orion, and NSDMiner to evaluate the performance of our system. We did not choose any of the three previous active approaches (Pinpoint, X-Trace, and ADD), because all these approaches are application-dependent. In other words, they either require modification of the applications (Pinpoint and X-Trace) under-study, or need to know details about the application protocols (ADD).

11. DISCUSSION

In this section, we discuss security applications of *Rippler* and clarify the differences between dependency detection and other applications of network watermarking.

11.1 Security Use Scenarios for Dependency Information

The results of our analysis can be used in different scenarios to improve the system security: for example, by correlating security events, prioritizing security events, prioritizing network assets, understanding underlying service dependencies, debugging network topology, improving service efficiency (by co-locating depending services), or localizing distributed system bugs.

The high volume of the alerts generated by intrusion detection systems (IDS) makes it difficult for security administrators to process this information. One way to address this challenge is to summarize and prioritize the alerts generated by different sensors. The dependency information about the servers can be used for this purpose.

Designing the network topology for a large network is complicated and error-prone. An administrator can easily compromise the security and efficiency of the system by misplacing the depending services. Dependency detection approaches can be used to fix topology bugs in the network structure. For example, an administrator may place an important service A (DB service) into a high-security network zone, while another service B, that service A depends on, is placed in an unsafe zone. Dependency detection can help the administrators in improving the security and efficiency of the system by placing these services in the appropriate zones.

Finally, distributed systems are often complex systems and comparatively susceptible to errors. Locating the errors in a distributed system is a challenging task. Dependency detection approaches can be used to limit the possible paths that could have led to the failure of a service.

11.2 Watermarking for Deanonymization

Watermarking has been widely used to deanonymize connections in anonymous communication systems [17, 18]. It has also been used in tracing watermarked attack/stepping-stone flows [15, 19, 20]. An anonymous communication network typically encrypts the incoming connections and (after repacketizing or other procedures to make deanonymization difficult) forwards them through a semi-randomly selected path of network nodes, and, at the end, it delivers the packets to the destination. Deanonymization of these connections is a serious threat to the security and privacy of the users of these anonymous communication networks.

Although watermarking has been widely used for deanonymization purposes, applying the same techniques for detecting service dependencies is far from trivial. The reason is that deanonymization methods match pairs of connection carrying the same information to each other, while dependency detection systems match services that their connections are causally related, and not usually carrying the same information.

11.3 Limitations

Rippler needs to be able to delay the traffic to the servers under-study at specific times. This requires the delayer component to be placed between the services and their corresponding clients. In other words, *Rippler* cannot verify or reject dependencies between two services if it is not able to delay requests to either of them.

Another limitation of *Rippler* is that because each service gets delayed in specific time windows, *Rippler* should know the set of under-study services beforehand.

One more limitation of *Rippler* is that for each particular service, only a small fraction of all the requests to that service are delayed[8], and, therefore, if a service does not have a large enough number of requests, the chances that a big enough number of its requests are delayed are small, and its dependencies would not be detectable. This problem can be solved by allocating a larger number of delaying time slots to services with less traffic.

12. CONCLUSIONS

In this paper, we presented a new application-independent active approach (*Rippler*) to detect dependencies among services using traffic watermarking. We showed, both analytically and experimentally, that *Rippler* can achieve arbitrarily low false positives if provided with large enough data sets. We compared *Rippler* with previous dependency-detection systems using a set of general effectiveness criteria for dependency-detection systems. Furthermore, we applied *Rippler* to a real-world network, and compared its results with three previous systems and showed that *Rippler* outperformed those systems.

13. REFERENCES

[1] S. Bagchi, G. Kar, and J. Hellerstein. Dependency analysis in distributed systems using fault injection: Application to problem determination in an e-commerce environment. In *In Proc. 12th Intl. Workshop on Distributed Systems: Operations & Management*, 2001.

[2] P. Bahl, P. Barham, R. Black, R. Ch, M. Goldszmidt, R. Isaacs, S. K, L. Li, J. Maccormick, D. A. Maltz, R. Mortier, M. Wawrzoniak, and M. Zhang. Discovering dependencies for network management. In *In Proc. V HotNets Workshop*, 2006.

[3] 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. *SIGCOMM Comput. Commun. Rev.*, 37, 2007.

[4] A. Brown, G. Kar, G. Kar, and A. Keller. An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. In *In Seventh IFIP/IEEE International Symposium on Integrated Network Management*, 2001.

[5] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Management. In *Proceedings of NSDI'04*, 2004.

[6] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl. Automating network application dependency discovery: experiences, limitations, and new solutions. USENIX Association, 2008.

[7] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), Oct. 2004.

[8] T. Coladarci, C. Cobb, E. Minium, and R. Clarke. *Fundamentals of Statistical Reasoning in Education*. Wiley/Jossey-Bass Education. John Wiley & Sons, 2010.

[9] D. Dechouniotis, X. Dimitropoulos, A. Kind, and S. Denazis. Dependency detection using a fuzzy engine. In *Proceedings of the Distributed systems: operations and management 18th IFIP/IEEE international conference on Managing virtualization of networks and services*, DSOM'07, pages 110–121. Springer-Verlag, 2007.

[10] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *In NSDI*, 2007.

[11] S. Kandula, R. Chandra, and D. Katabi. What's going on?: learning communication rules in edge networks. *SIGCOMM Comput. Commun. Rev.*, 2008.

[12] A. Kind, D. Gantenbein, and H. Etoh. Relationship discovery with netflow to enable business-driven it management. In *Business-Driven IT Management, 2006. BDIM '06. The First IEEE/IFIP International Workshop on*, pages 63 – 70, april 2006.

[13] A. Natarajan, P. Ning, Y. Liu, S. Jajodia, and S. E. Hutchinson. NSDMiner: Automated Discovery of Network Service Dependencies. In *In proceedings of IEEE International Conference on Computer Communications (INFOCOM '12)*, March 2012.

[14] L. Popa, B. gon Chun, J. Chandrashekar, N. Taft, and I. Stoica. Macroscope: End-Point Approach to Networked Application Dependency Discovery, 2009.

[15] Y. J. Pyun, Y. Park, D. S. Reeves, X. Wang, and P. Ning. Interval-based flow watermarking for tracing interactive traffic. *Comput. Netw.*, 56(5):1646–1665, Mar. 2012.

[16] O. Thonnard, L. Bilge, G. O'Gorman, S. Kiernan, and M. Lee. Industrial espionage and targeted attacks: Understanding the characteristics of an escalating threat. In D. Balzarotti, S. J. Stolfo, and M. Cova, editors, *RAID*, volume 7462 of *Lecture Notes in Computer Science*, pages 64–85. Springer, 2012.

[17] X. Wang, S. Chen, and S. Jajodia. Tracking anonymous peer-to-peer voip calls on the internet. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 81–91, New York, NY, USA, 2005. ACM.

[18] X. Wang, S. Chen, and S. Jajodia. Network flow watermarking attack on low-latency anonymous communication systems. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 116 –130, may 2007.

[19] X. Wang, J. Luo, and M. Yang. An efficient sequential watermark detection model for tracing network attack flows. In *Computer Supported Cooperative Work in Design (CSCWD), 2012 IEEE 16th International Conference on*, pages 236 –243, may 2012.

[20] X. Wang and D. S. Reeves. Robust correlation of encrypted attack traffic through stepping stones by manipulation of interpacket delays. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 20–29. ACM, 2003.

[8]The reason for this is to prevent delays from different services to affect other under-study services