On the Accurate Identification of Network Service Dependencies in Distributed Systems

Barry Peddycord III, Peng Ning
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
Emails: {bwpeddyc, pning}@ncsu.edu

Sushil Jajodia
Center for secure Information Systems
George Mason University
Fairfax, VA 22030
Email: jajodia@gmu.edu

Abstract

The automated identification of network service dependencies remains a challenging problem in the administration of large distributed systems. Advances in developing solutions for this problem have immediate and tangible benefits to operators in the field. When the dependencies of the services in a network are better-understood, planning for and responding to system failures becomes more efficient, minimizing downtime and managing resources more effectively.

This paper introduces three novel techniques to assist in the automatic identification of network service dependencies through passively monitoring and analyzing network traffic, including a logarithm-based ranking scheme aimed at more accurate detection of network service dependencies with lower false positives, an inference technique for identifying the dependencies involving infrequently used network services, and an approach for automated discovery of clusters of network services configured for load balancing or backup purposes. This paper also presents the experimental evaluation of these techniques using real-world traffic collected from a production network. The experimental results demonstrate that these techniques advance the state of the art in automated detection and inference of network service dependencies.

Tags: network services, dependency, service clusters, research

1 Introduction

Enterprise networks are large and complicated installations that feature intricate and often subtle relationships between the individual nodes in the system. They are usually composed of multiple *network services* that provide the core functionalities to the rest of the network, such as authentication and database management services. Many services deployed in a network depend on additional services hosted in the same or other networks. Should a service go down, for example, due to a system failure, those that depend on it will also be affected. Understanding the dependencies of mission-critical services in distributed systems enables system administrators to better prepare for and respond to network failures and attacks [9].

For example, a typical online banking application usually adopts a user front-end hosted on a web server, which depends on a database service that stores user account information as well as an authentication service that verifies the users who would like to use the online banking application. If the database or the authentication service goes down, the end users will not able to use the online banking application. Knowing the dependencies between these services will enable a system administrator to quickly diagnose the problem and recover the online banking application when system failures occur.

Production networks, however, are seldom this simple, and identifying the dependencies between network services in practice is a non-trivial task. Besides being large and complex, a production network is also dynamic, as nodes are added, removed, and repurposed within the network. In a large enterprise network, it is difficult to derive network service dependencies by only relying on the understanding of the network by experienced administrators. To address this problem, several techniques and tools have been developed to help system administrators identify and monitor network service dependencies efficiently and automatically.

1.1 Related Work

A naive approach to discovering network service dependencies is to manually investigate the configuration files of network services [12]. However, this is time-consuming, error-prone, and unable to keep up with changes in the network configuration. Researchers have proposed several automated approaches to determine such dependencies by analyzing the network traffic between network services. These approaches can be divided into *host-based* and *network-based* approaches.

Host-based approaches work by installing an agent on the nodes in the network to monitor the behavior of the processes running on it. Macroscope, for example, uses an agent that monitors the network interface to identify the processes that produce traffic as well as the dependencies between network services [12]. Similarly, Magpie tracks the execution paths of applications within a computer and follows them across the network to discover dependencies [4]. Pinpoint monitors the traffic path within the distributed system [5], while X-trace uses this traffic to develop trees of events required to execute a task. Constellation leverages distributed installation of agents and distributed machine learning to identify dependencies [3].

While these approaches are often accurate and effective, system administrators generally prefer to avoid installing extra software in their production environments due to concerns regarding security, performance, or administrative overhead, which poses a barrier to their adoption. Recent efforts have instead explored *network-based* approaches that treat each host as a black box and passively analyze the network traffic between them.

Sherlock, for example, identifies dependencies based on the frequency of co-occurrences of network traffic within a small time window [2]. eXpose further utilizes Sherlock's intuition, using statistical measures to filter out traffic that is less likely to lead to dependencies [8]. Orion identifies dependencies using the delay distribution of network traffic [6]. Another approach applies fuzzy-logic algorithms to classify dependencies by building an inference engine from the traffic [7]. NSDMiner uses nested network traffic flows to identify dependencies with similar detection rates but much lower false positives than the earlier network-based approaches [10].

In spite of these earlier efforts, network-based approaches to discovering network service dependencies are still susceptible to false positives and false negatives. Without access to application-level information about the traffic, it is difficult to separate which traffic is caused by a dependency relationship and which traffic is simply coincidental. In addition, because the dependencies are extracted entirely from network traffic, the dependencies of infrequently used services can be missed due to lack of data. It is highly desirable to have more effective approaches that can also identify the dependencies involving infrequently used network services.

1.2 Our Contribution

In this paper, we develop several novel techniques to address the above problems, which are then used to extend NSDMiner [10], the most recent addition to the network based approaches for the discovery of network service dependencies. While we have developed and released an open source implementation of NSDMiner [11] concurrently with this paper, the algorithms discussed are applicable to any implementation.

Firstly, we provide a new ranking mechanic that can more accurately identify true network service dependencies. Note that all dependency candidates inferred from network traffic could be due to either true network service dependencies or coincidental network activities. In this paper, we introduce a new logarithm-based ranking scheme, which considers not only the correlation between related network flows, but also the implication of coincidental network traffic. As a result, this ranking scheme can rank dependency candidates more accurately and consistently. In other words, compared with NSDMiner, it is more likely for the new scheme to rank the dependency candidates due to true network service dependencies higher than false ones.

Secondly, we develop a novel technique to infer network service dependencies for which very little explicit traffic exists. There are infrequently used network services, which do not produce enough traffic to allow their dependencies to be inferred accurately. We observe that many network services, though possibly deployed for different purposes, share similar configurations. For example, the same Apache HTTP server software may be used (on two different servers) to host both a popular website and an infrequently visited academic web server. Despite the separate installations and different purposes, these two services and the network services supporting them are likely to share some similarities. Leveraging this observation, we develop a technique that attempts to identify groups of similar services, extract the common dependencies of members of the group, and then infer the dependencies of the individual services, particularly those involving infrequently used services, from the common dependencies.

Finally, we introduce a new approach to identify redundant *service clusters* in a network. A service cluster consists of a set of identical network services, which is often used for load-balancing or fault-tolerance purposes. As pointed out in [6, 10], service clusters introduce additional challenges to the network-based automatic discovery of service dependencies, since different service instances will split the network traffic intended for getting certain services and thus make the dependency inference more difficult. In this paper, we develop a new way to automatically infer service clusters. This approach is inspired by an observation of load-balancing service clusters, i.e., services in a service cluster are often accessed with similar frequencies. We thus develop an approach to identify service clusters by observing what services are consistently contacted in groups.

We have implemented these techniques as extensions to NSDMiner, a tool that has been demonstrated to be promising for automated discovery of network service dependencies recently [10]. We have performed experimental evaluation with network traffic collected from our department network. The experimental results demonstrate that our new ranking scheme can further reduce the false positive rate compared with NSDMiner with comparable detection rate. In addition, our approach for inferring network service dependencies can further reduce the false negative rate, and our approach for identifying service clusters is able to identify most of the service clusters in our network.

The rest of this paper is organized as follows: Section 2 presents some preliminary discussion of network services and service dependencies, as well as a brief introduction to NSDMiner. Section 3 describes the proposed techniques in detail. Section 4 discusses our experimental results. Section 5 concludes this paper and points out some future research directions. Following the paper, we provide an appendix that highlights implementation concerns and best practices for operators who may wish to use NSDMiner in their own networks.

2 Preliminaries

2.1 Network Services and Network Service Dependencies

A *network service* is a process running on a computer in a network that listens on a port for connections from client processes running on other computers, and provides a service over the network. A network service can be represented by a triple $(ip, port, protocol)$, which specifies the machine the service is hosted on, the type of application, and the network protocol (TCP or UDP). Intuitively, given two network services A and B, *A depends on B* if A accesses resources on B, either indirectly or directly, to complete its own tasks. Such a dependency is denoted $A \to B$, where A is referred to as the *depending* service and B is referred to as the *depended* service [6]. (Some previous work refers to the depending service and the depended service as the *dependent* service and the *antecedent* service, respectively [9].)

Network service dependencies fall into two categories based on whether A accesses B's resources indirectly or directly, referred to as *remote-remote* and *local-remote* dependencies, respectively [6]. A remote-remote dependency $A \to B$ is a dependency where a client cannot access A until it has first accessed B. For example, many web services depend on the DNS service. In order for a client to access a web server via its host name, it usually needs to contact a DNS server to translate the web server's host name to its IP address.

A local-remote dependency $A \to B$ is a dependency relationship where A directly contacts B during the course of its operation. In production networks, this is a much more common dependency relationship [10]. Indeed, production networks typically expose a few public applications or services, which are supported by many internal, unexposed services that provide the core functionality. This paper focuses on local-remote dependencies.

It is also possible for one network service to depend on another service running on the same host, or for dependencies to exist between virtual machines on a single host. In this case, the communication will be done over a channel such as the loopback interface [1]. However, such dependencies do not produce any visible network traffic, and cannot be detected by passively observing the traffic. Our approach is therefore directed at inter-host network service dependencies.

2.2 The NSDMiner Approach

We first provide a brief overview of NSDMinder [10] below, since our techniques are developed as its extensions.

NSDMiner takes a list of *network flows* as its input. A network flow is a continuous, directed stream of packets between two processes running on different hosts. Each network flow is represented as a tuple of seven attributes *(StartTime, EndTime, SourceIP, SourcePort, Protocol, DestIP, DestPort)*. NSDminer mainly considers TCP flows and UDP flows. A TCP flow represents the traffic of a TCP session, and can be identified by locating the 3-way handshake and 4-way handshake or reset that signal the beginning and the end of the session. A UDP flow between two services consists of a continuous stream of UDP packets where the delay between two consecutive packets is no greater than a certain threshold.

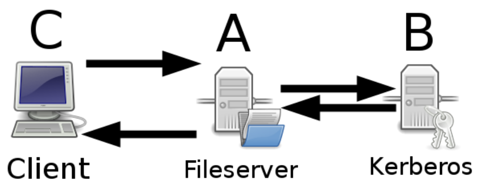

Figure 1: Nested network flows when $A \to B$.

NSDMiner utilizes a common pattern observed in network applications, illustrated in Figure 1. In general, when a client C accesses a network service A, the network flow between the two will usually be open for the entire duration of the client session. If the network service depends on another service B, it usually contacts B while the client session is running, producing a flow between A and B, nested within the flow from C to A.

To harness this observation, for each observed flow from *C* to *A*, NSDMiner counts the number of flows originating from *A* and occurring during the flow from *C* to *A*. NSDMiner models this interaction in a *communication graph*, where each node represents a network service with a weight identifying how many times it occurs as the destination of a flow, and each directed edge represents a *dependency candidate* between two network services with a weight indicating the number of times the nested behavior occurs.

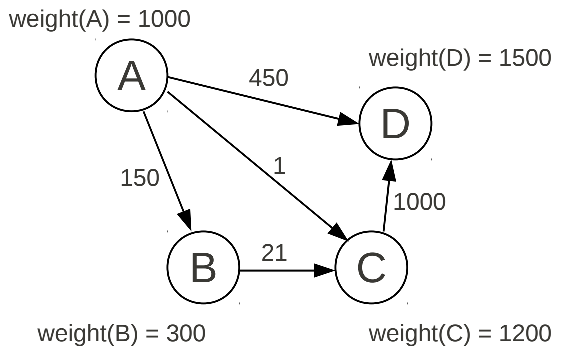

Figure 2: A communication graph produced by NSDMiner.

Figure 2 shows a communication graph produced by NSDMiner where *A*, *B*, *C*, and *D* are all services running in a production network. Take nodes *A* and *B* and the edge $A \to B$ as an example. This indicates that the network service *A* has been accessed 1,000 times, and *A* further used service *B* 150 times out of these accesses. *B* was accessed 300 times, 150 of of which come from *A* and another 150 coming from client machines which are not represented in the graph.

Because multiple network services can be hosted on the same physical machine, it is possible that a flow from *A* to *B* may occur during multiple overlapping client sessions, thus creating ambiguity in deriving the weights in the communication graph. NSDMiner uses two modes to mitigate this ambiguity: the *exclusive mode*, where overlapping client sessions are simply discarded, and the *shared mode*, where each of the overlapping sessions is given an equal fraction of a count.

Given a dependency candidate $A \to B$, the ratio $\frac{weight(A \to B)}{weight(A)}$ is used as the *confidence* of this candidate. Intuitively, the higher this confidence is, the more likely it is for *A* to depend on *B*. NSDMiner directly uses the ratio-based confidence to prune false dependencies due to coincidental traffic; any dependency candidates whose confidence falls below a user-specified threshold will be pruned from the graph. A lower threshold will have greater coverage of true dependencies, but will also introduce potentially more false positives. For the sake of presentation, we call this approach the ratio-based (confidence) ranking scheme.

3 New Techniques for Discovering Network Service Dependencies

In this section, we give the details of the proposed techniques. We first present a logarithm-based ranking scheme to assign confidence values to dependency candidates and rank them more appropriately. This scheme enables us to further reduce false positives beyond the original NSDMiner. We then describe our approach for identifying dependencies of network services for which very little traffic exists, improving the overall detection rate. Finally, we introduce an approach towards automatically discovering clusters of network services, which allows us to aggregate the output of NSDMiner to more effectively identify network service dependencies.

3.1 Logarithm-Based Ranking

Let us first understand the limitations of the ratio-based ranking scheme in the original NSDMiner with the assistance of Figure 3, which shows the confidence of each dependency candidate in Figure 2 as calculated by the ratio-based ranking scheme.

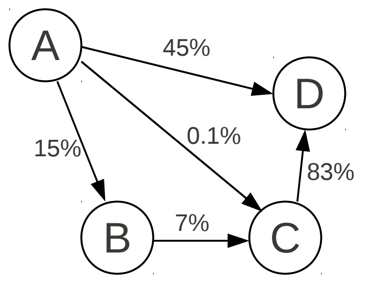

Figure 3: Ratio-based ranking for the graph in Figure 2.

The ratio-based ranking scheme has two major limitations. First of all, it does not take the magnitude of the weights into consideration when assigning confidence values. Consider the dependency candidate $A \to D$, where *A* accesses *D* almost every other time when *A* itself is accessed. This is very consistent behavior out of the 1,000 accesses to *A*, which is more likely to be caused by a dependency relationship than coincidental traffic. Note that it only has a confidence value of 45%. Imagine that additional traffic is collected, $weight(A)$ increases to 10,000, and $weight(A \to D)$ increases to 4,500. Note that the confidence still remains at 45%, though intuition suggests that it would be even more unlikely for such a high number to be produced by coincidence. Because a raw ratio is used, the magnitude of the two weights have no effect on the final confidence value of the dependency candidate.

Secondly, the ratio-based ranking assumes that the confidence in a dependency relationship scales linearly with the weight of the edge. In other words, it assumes that each observation of a nested connection in support of a dependency candidate contributes equally to its confidence value. However, after a certain threshold, it is no longer reasonable to assume that additional observations are caused by coincidence. At this point, the dependency candidate should have already been assigned a high confidence value. For example, consider Figure 2, where $weight(A) = 1,000$, $weight(A \rightarrow C) = 1$, $weight(C) = 1,200$, and $weight(C \rightarrow D) = 1,000$. Intuitively, 100 new observations in support of $C \rightarrow D$ would have almost no effect on the confidence of $C \rightarrow D$, while the same number of observations for $A \rightarrow C$ would have much more impact on the confidence of $A \rightarrow C$.

These two issues suggest that the ratio-based ranking scheme does not provide the best estimate of the probability of a dependency candidate being true. We propose to adopt a *logarithm-based* ranking mechanic that addresses these issues. Specifically, we propose to define the *confidence of a dependency candidate* $A \rightarrow B$ as $log_{weight(A)} weight(A \rightarrow B)$. Figure 4 shows the logarithm-based confidence values for the same scenario in Figure 2. Note the changes in confidence values in comparison with Figure 3.

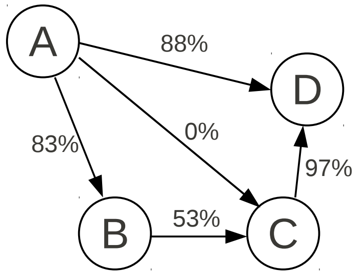

Figure 4: Logarithm-based ranking for the graph in Figure 2.

Both of the above concerns are addressed by the following two properties of the logarithm function that hold for all $1 < a < b < c < d$:

- If $\frac{a}{c} = \frac{b}{d}$, then $log_a c < log_b d$;
- If $\frac{c}{d} - \frac{b}{d} = \frac{b}{d} - \frac{a}{d}$, then $log_d c - log_d b < log_d b - log_d a$.

The first property ensures that when the ratio between the weights of a node and the outgoing edge from the node remain constant, the dependency candidates with greater weights will be assigned higher confidence values than those with lower weights. This reflects the intuition that having more data removes the effect of random coincidence and that the confidence value given to a dependency candidate should take the amount of data collected into account. This has the effect of giving the more obvious dependency candidates higher confidence, preventing them from being filtered out when pruning candidates.

The second property helps reject the notion that the confidence of a dependency candidate scales linearly with the observed traffic, by giving more weights to earlier observations of dependency behavior, and slowly reducing the growth rate as more observations are made. With the ratio-based ranking scheme, both true and false dependency candidates are all clustered at very low confidence values, especially when large amounts of data are collected [10]. This requires the usage of very low filtering thresholds to avoid removing true positives from the results, which unfortunately will also retain a good number of false positives. The logarithm-based ranking scheme puts more disparity between these candidates, making true positives with less traffic more resistant to filtering. As a result, it improves the detection rate and at the same time reduces the false positive rate.

3.2 Inference of Service Dependencies

Our second contribution is an approach for inferring the dependencies of network services that are used infrequently and have insufficient observed traffic associated with them. While many services in a network are used regularly, some services are only accessed occasionally, or at least accessed infrequently during the data collection aimed for discovering network service dependencies. This introduces a great challenge to all automated approaches that attempt to discover network service dependencies by analyzing the activities of these services.

In the following, we develop an approach that uses additional properties in the network beyond network traffic to infer such obscure network service dependencies.

Our approach uses the overlap between the depended services of different network services to infer missing dependencies. When two depending services have several depended services in common, even if the confidence values associated with their respective dependency candidates are low, the overlap itself provides evidence that can be used to make inferences about poorly ranked or potentially missing true dependencies.

While it is not unusual for two network services to share some depended services by coincidence, it is highly unlikely that there will be a substantial overlap in the depended services between them if they are not related. We therefore use the amount of overlap to quantify the similarity between the two services. If a heavily used service and an infrequently used service have a large number of depended services in common, this is likely not the result of a coincidence. We harness this observation to boost the confidence values of the dependency candidates involving infrequently used services.

Examining the similarity between service dependencies using pairs of (depending and depended) services is sensitive to errors. To mitigate the problem, we instead identify groups of network services with large amounts of overlap in terms of their depended services. The members of these groups will have certain common depended services that most of them *agree* on. If most members of the group agree on a particular service as a depended service, we interpret this observation as evidence to infer that the other group members also depend on this service, and add new dependency candidates accordingly.

Given two network services A_1 and A_2, we define the *similarity* between these two services (w.r.t. their depended services) as $Sim(A_1, A_2) = \frac{|\mathscr{D}(A_1) \cap \mathscr{D}(A_2)|}{max(|\mathscr{D}(A_1)|, |\mathscr{D}(A_2)|)}$, where $\mathscr{D}(A_1)$ and $\mathscr{D}(A_2)$ are the sets of services that A_1 and A_2 depends on, respectively. Two services A_1 and A_2 are *similar* if (1) they use the same protocol with the same port and (2) $Sim(A_1, A_2)$ is greater than a threshold.

To model the similarity between services systematically, we build a *similarity graph* based on the communication graph produced by NSDMiner. The nodes in a similarity graph represent the network services, and an undirected edge exists between two nodes if and only if they are similar, as defined above. Figure 5 shows an example of a similarity graph.

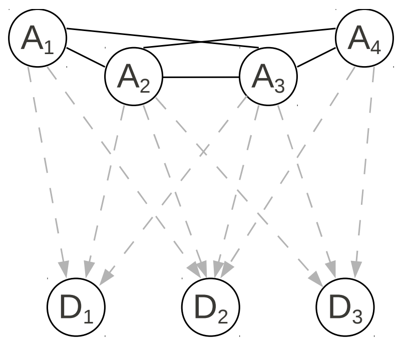

Figure 5: An example similarity graph (solid lines) overlaying on its communication graph, where the similarity and agreement thresholds equal 50%. A dashed line $A_i \rightarrow D_j$ represents that A_i potentially depends on D_j, and a solid line $A_i \rightarrow A_j$ represents that A_i and A_j are similar. The dependency candidates $A_1 \rightarrow D_3$ and $A_4 \rightarrow D_1$ will be inferred because D_1, D_2, D_3 are all agreed on by the group of services in A.

A group of similar network services is represented by a connected subgraph in the similarity graph. After identifying the services that are members of this group, we identify the common depended services that the members of the group agree on. The *agreement* on a depended service D is defined as the ratio between the number of services in the group that depend on D and the size of the group. If the agreement on D is greater than a threshold, we can infer that D is a common depended service of the group, and any members that do not depend on D will have such a dependency candidate inferred.

After identifying the common depended services of a group, we need to estimate the corresponding confidence values. To do so, we modify the original communication graph by adding edges to represent the newly inferred dependencies, and set the weights of the newly inferred dependency candidates to a value such that the ratio between the weight of the candidate and its depending service is proportional to the greatest ratio in the group.

Specifically, let $\mathscr{A}$ be a set of similar services $\{A_1, A_2, ..., A_n\}$, and $\mathscr{D}$ be the set of $\mathscr{A}$'s common depended services $\{D_1, D_2, ..., D_m\}$. For each pair of services A_i, D_j, if the dependency $A_i \rightarrow D_j$ does not exist, we add an edge from A_i to D_j. As a result, there is an edge from each member in $\mathscr{A}$ to each member in $\mathscr{D}$.

For each depended service D_j in $\mathscr{D}$, let A_{max} be the service in $\mathscr{A}$ such that the ratio $\frac{weight(A_{max} \rightarrow D_j)}{weight(A_{max})}$ is the greatest. For each member service A_i in $\mathscr{A}$, set the $weight(A_i \rightarrow D_j)$ as $\frac{weight(A_i) weight(A_{max} \rightarrow D_j)}{weight(A_{max})}$. The new communication graph can then be used with the logarithm-based ranking to calculate the final confidence value of each candidate.

When the weight of an edge in the extended communication graph is set to this proportional weight, the final confidence value of an inferred dependency will be proportional to how often the depending service is accessed. As a result, even when coincidental traffic causes false dependencies to be inferred, the confidence values of these inferred candidates are low enough to be pruned before it is reported as a dependency.

3.3 Discovery of Service Clusters

Finally we introduce an approach for identifying service clusters in a network. A network service cluster is a set of services $\{B_1, B_2, ..., B_n\}$ such that if a dependency $A \rightarrow B_1$ exists, B_1 can be substituted by any other service in the cluster without affecting the completion of A's task. These clusters typically provide load-balancing or backup for core services to improve performance and fault tolerance.

Both Orion [6] and NSDMiner [10] proposed to use the knowledge of service clusters to improve the detection of dependencies that involve service clusters. However, both approaches rely on human users to provide such information [6, 10]. Such manual interventions require additional human efforts, and are time-consuming and error-prone. In the following, we develop an automated approach to discovering service clusters.

Let us start with the observation behind the proposed approach. When a network service A_1 depends on services in a cluster $\{B_1, B_2, B_3\}$, it is highly likely that $A_1 \to B_1$, $A_1 \to B_2$, and $A_1 \to B_3$ will be observed as dependency candidates. In a load-balancing configuration, the services that depend on the cluster will access the members of the cluster with roughly equal probability. This means that over time, so long as the load-balancing assignments are not "sticky," all services that depend on the cluster will feature all of the members in their lists of depended services. In a backup configuration, the services that depend on the primary server $B_{primary}$ will feature it in their list of depended services in normal situations. When $B_{primary}$ fails, the backup service B_{backup} will take over the service by $B_{primary}$, and these depending services will now have B_{backup} as their depended services (along with $B_{primary}$ before the failure).

In both cases, when services consistently appear together in the lists of depended services of the same depending service, it is likely that they are part of service clusters. By counting the number of cases for which depended services appear together for the same depending service, we can estimate the likelihood that a cluster exists.

Now let us present the approach for discovering service clusters. Our approach takes a communication graph produced by NSDMiner as its input, disregarding the weights of the nodes and edges. The algorithm produces a *clustering graph*, where an edge exists between two nodes B_1 and B_2 if and only if (1) B_1 and B_2 use the same protocol with the same port and (2) there exists a service A such that $A \to B_1$ and $A \to B_2$ are dependency candidates. The weight of each edge, called the *support* for the pair of services, is the number of services for which B_1 and B_2 are both depended services. Figure 6 shows an example of a clustering graph generated from a communication graph.

This approach assumes that network services that are parts of clusters will occur together frequently in the lists of depended services, and that pairs of coincidental services will occur in the same list of depended services infrequently. After generating the clustering graph, all edges with weights less than a support threshold will be dropped, and the remaining connected subgraphs will be considered the service clusters of the network. Because clusters are expected to have high supports while the support of coincidental pairs of services will be low, only a moderate absolute support threshold should be necessary for identifying the clusters, such as a value between 5 and 10, depending on the size of the network.

Rather than using clusters to infer new candidates

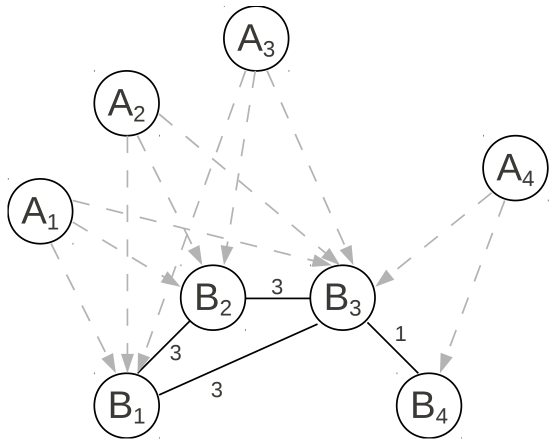

Figure 6: An example of a cluster graph (solid lines) overlaid with its communication graph (dashed lines). Assume the support threshold is 2. $\{B_1, B_2, B_3\}$ is a cluster, while B_4 is not considered a part of this cluster.

or increase the weights of edges, we instead aggregate the results of NSDMiner so that instead of reporting a separate dependency for each member of a cluster ($A_1 \to B_1, A_1 \to B_2, A_1 \to B_3$), we treat all of the members of the cluster as a single group of depended services ($A_1 \to B_{cluster(1,2,3)}$). In the example in Figure 6, the total number of dependencies after this aggregation is reduced from 12 to 5, making the output smaller and more manageable, reducing the time needed to validate the results. In our evaluation, we show that even when the thresholds for clusters are very strict, the number of dependencies is still greatly reduced.

4 Evaluation

We have implemented the proposed techniques as extensions to NSDMiner. To have a realistic evaluation of these techniques, we use real-world network traffic collected from the production network of Department of Computer Science at North Carolina State University. In the following, we first give a brief description of the evaluation data set, and then present the experimental results.

4.1 Evaluation Data Set

Our experimental evaluation uses the same data set that were used to evaluate the original NSDMiner [10]. In the following, to make this paper self-contained, we repeat the description of this data set.

We evaluate our proposed techniques using the network traffic collected from the production network on the second floor of the building where our department is located, including all the internal traffic across subnets in the building. This network consists of 14 switches hierarchically organized in two levels. Each computer in this network is connected to a switch via a wall port or a port on the switch directly. The top-level switches are then connected to a master building switch through a 10G multimode fiber, which facilitates communication with the campus network outside of the building.

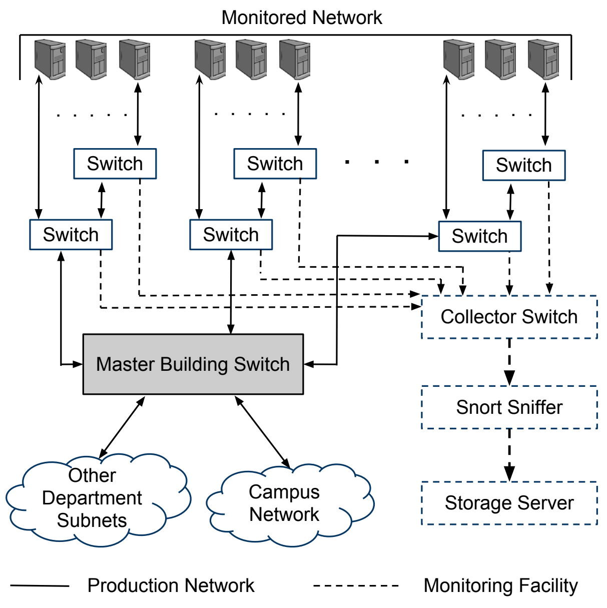

Figure 7: Diagram of the traffic monitoring facility [10].

Figure 7 shows a diagram of our traffic monitoring facility. In order to collect the traffic from our network, each of the switches is configured with a SPAN session so that packets that are routed throughout the building are also forwarded to a machine running a packet-sniffer. The packet-sniffer is Linux machine running *snort* [13] in packet-logging mode, where each packet is stripped down to its IP and TCP/UDP header information and forwarded to a secure data storage machine that is disconnected from the public Internet. We then convert these packet headers into network flow records using *softflowd* [14]. The final evaluation data set includes 46 days of network flows collected between 05/10/2011 and 06/25/2011, which consists of 378 million flow records.

The ground truth for this data set was generated with the assistance of the departmental IT staff. Our traffic covers a total of 26 servers, 3 of which are Windows machines, and 23 are Linux machines. These machines offer a total of 43 services possessing 108 dependencies. The majority of the Linux machines belong to individual faculty, and are used to provide shell and web services for their graduate students, while the rest provide infrastructure such as Kerberos, databases, and DFS replication.

Table 1 summarizes the services and their dependencies. Most of the services offered were dependent on DNS (53) for name resolution. Windows-based services were dependent on Active Directory (AD) services for domain queries (389) and authentication (88). The services that were dependent on those offered on dynamic ports (RPC) were also dependent on endpoint mapper to resolve the port numbers. Most of the Linux-based services were dependent on LDAP (389) for directory access. Two of the interesting services hosted were TFTP (69) and database (3306); they were running stand-alone and was not dependent on any other network service. Windows deployment service (WDS) and DFS replication service were offered on dynamically allocated ports and others were offered on standard well-known ports.

Table 1: Ground truth of service & dependencies

Service	**Instances**	**Dependencies**	
webservice (80, 443)	4	2	DNS, DBMS
webservice (80)	1	1	DNS
ssh (realm-4) (22)	5	2	Kerberos, DNS
ssh (realm-5) (22)	17	3	Kerberos, DNS, LDAP
svn (8443)	1	4	DNS, LDAP, port mapper, RPC
proxy DHCP (4011)	1	2	DNS, LDAP
DHCP (68)	1	1	DNS
email (25)	1	2	mail exchange server, DNS
endpoint mapper (135)	2	3	DNS, AD, Kerberos
WDS (RPC)	1	5	DNS, AD (LDAP, port mapper, RPC, Kerberos)
DFS replication (RPC)	2	5	DNS, AD (LDAP, port mapper, RPC, Kerberos)
SMB (445)	2	5	DNS, AD (LDAP, port mapper, RPC, Kerberos)
TFTP (69)	1	0	
database (3306)	2	0	

Note that even though we worked hard to identify all dependencies, there is a possibility that we might have missed some rare non-obvious ones. The reader is advised to keep this in mind while interpreting the experimental results.

4.2 Logarithm-Based Ranking

In order to evaluate the effectiveness of the logarithm-based ranking scheme, we run NSDMiner with our logarithm-based ranking scheme and compare the results to the original ratio-based scheme [10] for both shared and exclusive mode. We compare our results to the previous version of NSDMiner as it has already been demonstrated as the most effective network-based solution [10].

Table 2 shows an initial comparison of the true positive (TP), false positive (FP), and false negative (FN) for both ranking schemes. We choose a filtering threshold for the logarithm-based ranking scheme that set the number of true positives as close to those obtained by the ratio-based ranking at confidence value 0.5% as possible.

The dependency candidates with the highest false positive rates under the ratio-based ranking were those that were accessed the least. For example, in the ratio-based ranking scheme in the shared mode, Microsoft endpoint mapper (of which there are two instances, with three

Table 2: Comparing the false positive rates between the Ratio-based [10] and Log-based ranking schemes by setting the threshold to a value that keeps the number of true positives close to their value under the ratio approach.

	Shared Mode						Exclusive Mode					
	Ratio (0.5%)			Log (45%)			Ratio (0.5%)			Log (44%)		
Service	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
webservice	8	9	1	9	12	0	8	4	1	8	7	0
email	3	1	0	3	3	0	3	0	0	3	2	0
ssh(realm-4)	10	1	0	10	1	0	10	1	0	10	1	0
ssh(realm-5)	41	14	10	43	12	8	40	10	11	43	9	8
svn	4	1	0	4	0	0	4	1	0	4	0	0
proxy DHCP	2	1	0	2	0	0	2	1	0	2	0	0
DHCP	1	6	0	1	1	0	1	6	0	1	1	0
endpoint mapper	6	43	0	6	4	0	3	6	3	3	1	3
WDS (RPC)	4	10	0	3	1	1	4	1	0	2	0	2
DFS replication (RPC)	8	4	0	6	3	2	7	0	1	5	1	3
SMB	9	7	1	9	9	1	9	3	1	9	5	1
TFTP	-	1	-	-	1	-	-	0	-	-	0	-
database	-	1	-	-	2	-	-	1	-	-	2	-
Invalid Services	-	34	-	-	29	-	-	30	-	-	17	-
Total	96	133	12	96	78	12	91	64	17	90	46	18

dependencies each) reported a total of 43 false dependencies using 0.5% confidence [10]. One instance was accessed 465 times, while the other was accessed 133 times. In order for a candidate to be pruned at 0.5% confidence, it would need to have a weight of less than 2.3 and 0.6, respectively. In the latter case, a single nested flow was enough to consider the candidate as a true dependency. Most of the true positives (LDAP, Kerberos, and DNS) had weights above 10.

With the logarithm-based ranking scheme, all dependency candidates with weights less than or equal to 1 are immediately pruned with a confidence value of 0%, as a single observation is never convincing enough to consider it a true dependency. Because $log_{133}10$ is .47, the true positives in this example were given confidence values above 47%. Thus, in Table 2, the true positives are preserved and many false positives are properly pruned.

To see the overall performance of both the ratio-based and the logarithm-based ranking schemes, we plot the Receiver Operating Characteristic (ROC) curve in Figure 8. This figure shows the true positive rate as a function of the false positive rate for both ranking scheme in both the shared and the exclusive mode. We can see a substantial improvement in the detection rate for the same false positive rate when the logarithm-based ranking scheme is used. This is because we can filter more false positives at higher confidence thresholds without incorrectly pruning true dependencies. It is worth noting, however, that even with these improvements, the true positive rate still reaches its maximum value at the same point in both ratio-based and logarithm-based ranking schemes.

Figure 9 illustrates how the behavior of the confidence threshold differs between the ratio-based and the logarithm-based ranking schemes. In both shared and

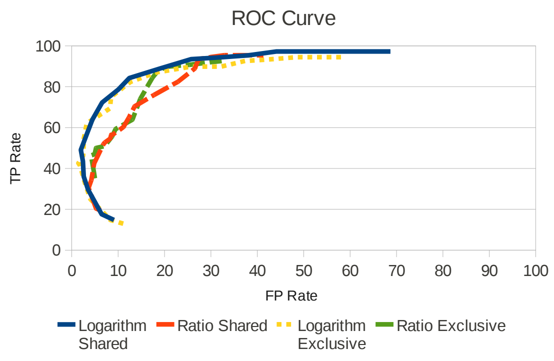

Figure 8: ROC curve for the logarithm-based and the ratio-based ranking schemes in both shared and exclusive modes.

exclusive modes with the ratio-based ranking, the true and false positive rates drop drastically as the confidence threshold increases. The false positive rate approaches its minimum value at a threshold of 10%, but at the cost of missing half of the true dependencies. In the logarithm-based ranking, both true and false positives are pruned at a slower rate. Due to the greater distance between their confidence values, true positives are pruned more slowly than false positives, so at the point when the false positive rate approaches its minimum value, the true positive rate is much higher than that under the ratio-based ranking.

In addition to improving the discovery of true dependencies with lower false positive rate, the logarithm-based ranking scheme more accurately predicts appropriate confidence values for dependency candidates. The confidence value given to a dependency candidate should roughly correspond to the probability that the candidate is true. For example, out of a set of candidates with

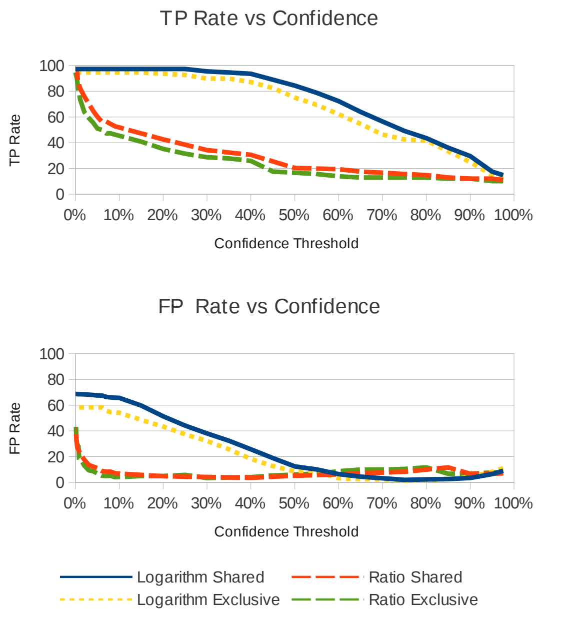

Figure 9: True/False positive rate v.s. confidence threshold.

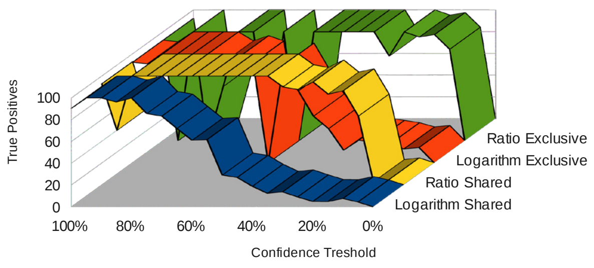

Figure 10: The percentage of true positives that are found at each particular ranking threshold. 95% in the 100 bin means that 95% of candidates reported with scores between 100 and 96 were true, while the other 5% were false. The ideal case is a straight line where $x = y$.

confidence values of 50%, half of them should be expected to be true positives. Figure 10 shows the distributions of true positives at 5% confidence intervals for both the logarithm-based and ratio-based ranking schemes in both shared and exclusive modes. Note that with the ratio-based ranking scheme, nearly all of the candidates ranked above 10% are true positives. As shown in Figure 9, this is because there are very few candidates ranked above 10% at all. As most of the results are clustered at low confidence values, it only takes a modest threshold to remove them from the results.

In contrast, and especially for the shared mode, the curve for the logarithm-based ranking scheme is much smoother and better illustrates how one would expect the true positive rates to map to the confidence rankings. When the confidence values are more meaningful, it is possible to make an informed decision about how many true and false positives should be expected when a certain cutoff value is chosen. While the curve for the exclusive mode is a bit choppier due to having a few high-confidence false positives, they still exhibit the same general patterns. Many false candidates are ruled out before reaching the pruning stage, and as a result, the true positive concentrations in exclusive mode are higher overall.

4.3 Inference of Service Dependencies

To test the effectiveness of the inference algorithm, we randomly choose from between 1 and 150 dependencies from the ground truth and remove all of the flows associated with those dependencies from the data set. We then run the inference algorithm to see how many of these dependencies could be recovered from the remaining data. We reason that if the inference algorithm can re-infer dependencies known to be true, then it should successfully be able to infer unknown dependencies in practice.

When running the inference algorithm, we use four combinations of high (75%) and low (25%) thresholds for similarity and agreement to determine the effect that each threshold has on the ability of the approach to recover missing dependencies. Only dependencies with the (logarithm-based) confidence values greater than 30% are used for these experiments. In the following, we only discuss the results of the shared mode in detail; the results of the exclusive mode are similar. For each pair of similarity and agreement thresholds, we ran 10,000 trials of removal and recovery.

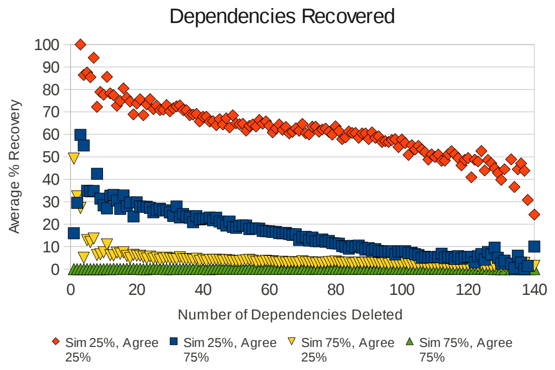

Figure 11: Dependencies recovered where all candidates with the logarithm-based confidence values less than 30% are removed. We show the average percentage of the recovered dependencies as a function of the number of removed dependencies. Each color/shape represents a certain similarity-agreement pair of thresholds.

Figure 11 shows the average percentage of recovered dependencies as a function of the number of dependencies that were removed. As one would expect, as more traffic is removed, it becomes harder to infer dependencies from what is left.

Increasing the agreement threshold has a much more dramatic effect on the recovery rate than the similarity threshold. When both the similarity and agreement thresholds are set to low values such as 25%, it is possible to recover many dependencies, and when both thresholds set to high values such as 75%, no dependencies are recovered at all.

In addition to recovering missing dependencies effectively, the inference approach is able to do so without inferring false positives. Figure 12 shows the average number of false positives that were inferred when running the previous experiments, while Figure 13 demonstrates the overall improvement to the ROC curve. Not only do we reach 100% coverage, we do so with a relatively low false positive rate.

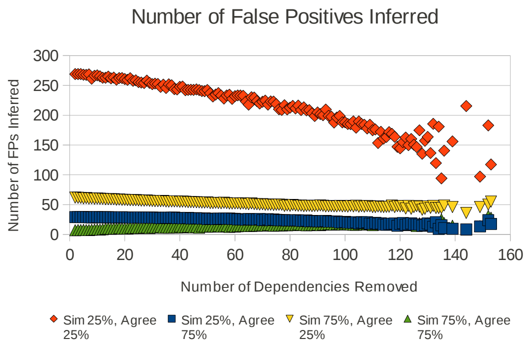

Figure 12: This shows the number of false positives inferred during the same experiments described earlier. Note that while there are often several false positives inferred, many of them are can be pruned in later steps of NSDMiner as they often have low confidence values.

In order to understand the effects of these thresholds, we discuss the specific behavior for our largest group of similar machines, our SSH services. These services mostly correspond to individual faculty machines that give shell and website services to their graduate students. By their nature, some of these are less frequently used than others.

With similarity thresholds of 25%, 27 such services, covering all true cases and 5 false cases, were considered similar. When the agreement threshold was low at 25%, all three true dependencies (and two false positives) were found to be in common and inferred as dependencies to the rest of the group, contributing to the full coverage of the ground truth. At 75% agreement, the only service that could be inferred was DNS.

When the similarity threshold is set to 75%, this group is reduced to seven members. When there are less members, it is easier to reach 25% agreement on depended services, meaning that four additional false positives, were inferred as a result. At 75% agreement, the only common depended services were Kerberos and DNS.

While our experiments are performed on a real-world network, it is not necessarily representative of networks in general. Additional work on more diverse configurations will be needed to identify the most appropriate choices for similarity and agreement thresholds. One important factor in this choice is the number of service clusters that are used in the network. The SSH services depended on three of the same clusters, giving them up to 13 true depended services in common, making it possible for them to have dependencies inferred with thresholds up to 50%. If there are fewer clusters in the network, then lower thresholds will be needed to find similar services and common dependencies.

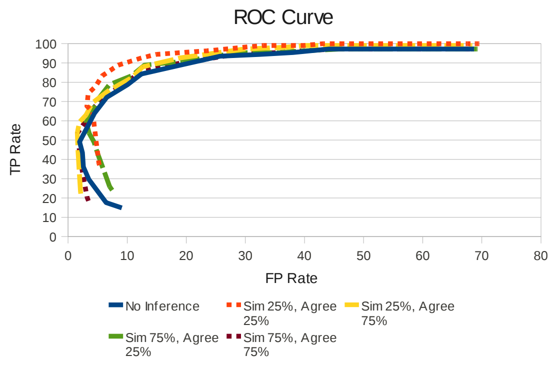

Figure 13: ROC curve for various similarity-agreement thresholds. A higher agreement threshold prevents the eager acceptance of false positives, however it still raises the confidence of true positives, allowing a higher confidence threshold to be set.

4.4 Discovery of Service Clusters

In order to evaluate the effectiveness of our clustering approach, we run our tool on the output of NSDMiner and compare the clusters that it reports to the clusters in our ground truth. There are two variables that affect the clusters that are reported: the support threshold, which specifies the number of depending services for which members of a cluster must appear together as depended services, and the confidence threshold, which sets the minimum confidence for a depended service for it to be counted as part of a cluster.

Our network contains a total of 10 service clusters, most of which contain 4–6 services. We consider a cluster reported by our clustering tool to be a true positive cluster so long as it contains at least two services where none of the reported services are put into the wrong cluster. Figure 14 shows the number of true clusters as a function of the number of clusters that are reported in total.

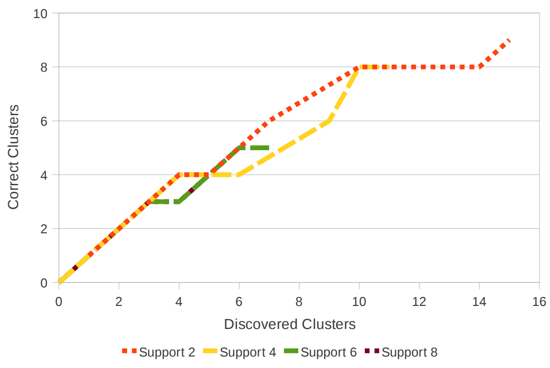

Figure 14: Quality of the clusters as a function of the number discovered. We consider a true cluster to be any size cluster that does not contain false positives. Therefore, a true cluster may be missing some of its members. All members are usually recovered when the confidence threshold is set below 60%.

We can see here that the clusters reported are fairly accurate. When 10 clusters are reported, clustering with low support values of 2 and 4 both report 8 true clusters. When looking carefully at the clusters reported, there was only one cluster that actually accepted false positives into its membership was Microsoft endpoint mapper. Services in the two distinct kerberos clusters were never mixed together.

The most difficult cluster to detect was our University's NTP time servers. No network services in our network explicitly depend on NTP, meaning that traffic to NTP was coincidental and confidence values associated with NTP dependencies were rightfully low. Due to these low thresholds, the NTP cluster could only be discovered at a support threshold of 2 and a confidence value below 30%.

An interesting set of false positives that survived through very high support thresholds were a number of web servers accessed over HTTP and HTTPS. Even at a threshold of 8, the HTTPS servers were still discovered with higher confidence than many of the true positives. SSH is interesting in that if a user uses their SSH account as a proxy for their web browser, then any web servers that are accessed will be considered dependency candidates of that SSH session. If popular websites are used frequently, such as Google (which appeared in a cluster of services running on port 80), they will considered clusters by this algorithm.

Overall, the quality of clusters reported was good. Even at low support thresholds, most of the clusters reported were correct, though this may be a property of our small network. The support threshold needs to be scaled to the size of the network, and the effect on larger networks still needs to be explored.

The value of clustering with respect to identifying dependencies comes from aggregating the services that make up a service cluster into a single depended service. Any degree of clustering, whether it is strict or lenient, will remove a large number of candidates from the output of NSDMiner. In most cases, this is because the true positives have the greatest tendency to influence the clusters that are formed. True positive clusters, by definition, are those with the greatest support values, and the support value of a cluster relates to how many depending services will benefit from the aggregation. So long as the clusters identified are accurate, as we have shown, then reducing the size of the output will be beneficial with no loss of information.

It is difficult to represent this benefit in the form of an ROC curve because aggregation actually increases the false positive rate by removing true positives from the output. Instead, we represent the benefit by showing the size of the output of NSDMiner as a function of the confidence thresholds and support thresholds used in Figure 15. In most cases, we can see an reduction in output size from anywhere between 25% to 50%.

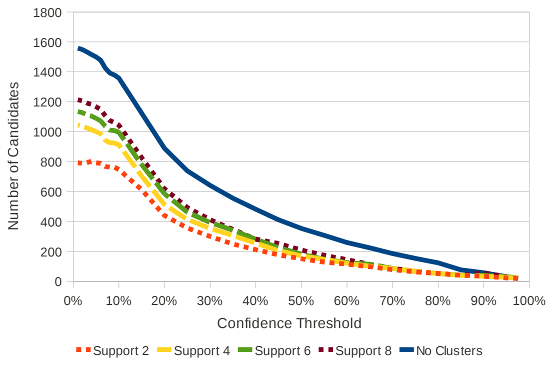

Figure 15: Reduction in the number of candidates for various support thresholds. The confidence threshold corresponds to the logarithm-based ranking introduced in this paper.

5 Conclusions and Future Work

In this paper, we introduced three techniques for improving the discovery of network service dependencies, and implemented them as modifications to the tool NSDMiner [10]. We developed a new ranking formula that makes it easier to judge whether or not a dependency candidate is true or false, an approach for inferring the dependencies infrequently-used services that are similar to others, and an approach for finding service clusters such as backup or load-balancing nodes in a network. Furthermore, we evaluated our approaches on production traffic in a university network, showing a substantial improvement in both true and false positive rates.

A number of issues are still left unexplored. For example, remote-remote dependencies are still not considered by NSDMiner at all. NSDMiner also relies on the assumption that network sockets are opened and closed on demand, meaning that it will miss dependencies involving middleware applications that keep connections open for extended durations. Finally, NSDMiner assumes that during the time that flows are collected, no changes occur in the network configuration, meaning that it is not able to keep up with a dynamically evolving network. Future work will address these issues and involve the collection additional data from a larger network to further test the effectiveness of these approaches.

Acknowledgements

We would like to offer our thanks to the continued support of our departmental IT staff, Carlos Benavente, Vadim Kuznetsov, Trey Murdoch, and Jeremy Meeler for their assistance in keeping our ground truth up-to-date, as well as Neal McCorkle and William Brockelsby for their help in managing our traffic monitoring infrastructure throughout our experiments. We would like to thank Arun Natarajan, who developed the first version of NSDMiner [10], for all the help and support with the original NSDMiner prototype and validation scripts. Finally, we would like to thank the conference reviewers and shepherds for their feedback in finalizing this paper for publication in LISA '12.

This work is supported by the U.S. Army Research Office (ARO) under a MURI grant W911NF-09-1-0525.

References

[1] BAHL, P., BARHAM, P., BLACK, R., CH, R., GOLDSZMIDT, M., ISAACS, R., K, S., LI, L., MACCORMICK, J., MALTZ, D. A., MORTIER, R., WAWRZONIAK, M., AND ZHANG, M. Discovering dependencies for network management. In *In Proc. V HotNets Workshop* (2006).

[2] BAHL, P., CHANDRA, R., GREENBERG, A., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2007), SIGCOMM '07, ACM, pp. 13–24.

[3] BARHAM, P., BLACK, R., GOLDSZMIDT, M., ISAACS, R., MACCORMICK, J., MORTIER, R., AND SIMMA, A. Constellation: automated discovery of service and host dependencies in networked systems. *MSR-TR-2008-67* (2008).

[4] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), USENIX Association, pp. 18–18.

[5] CHEN, M. Y., ACCARDI, A., KICIMAN, E., LLOYD, J., PATTERSON, D., FOX, A., AND BREWER, E. Path-Based Failure and Evolution Management. In *IN PROCEEDINGS OF THE INTERNATIONAL SYMPOSIUM ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION (NSDI'04* (2004), pp. 309–322.

[6] CHEN, X., ZHANG, M., MAO, Z. M., AND BAHL, P. Automating network application dependency discovery: experiences, limitations, and new solutions. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 117–130.

[7] DECHOUNIOTIS, D., DIMITROPOULOS, X., KIND, A., AND DENAZIS, S. Dependency detection using a fuzzy engine. In *Managing Virtualization of Networks and Services*, A. Clemm, L. Granville, and R. Stadler, Eds., vol. 4785 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007, pp. 110–121. 10.1007/978-3-540-75694-1_10.

[8] KANDULA, S., CHANDRA, R., AND KATABI, D. What's going on?: learning communication rules in edge networks. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication* (New York, NY, USA, 2008), SIGCOMM '08, ACM, pp. 87–98.

[9] KELLER, A., KELLER, E., BLUMENTHAL, U., AND KAR, G. Classification and computation of dependencies for distributed management. In *Proceedings of the Fifth International Conference on Computers and Communications (ISCC* (2000).

[10] NATARAJAN, A., NING, P., LIU, Y., JAJODIA, S., AND HUTCHINSON, S. E. NSDMine: Automated Discovery of Network Service Dependencies. *Proc. IEEE INFOCOM'11 In Submission* (2011).

[11] NSDMINER. http://sourceforge.net/projects/nsdminer.

[12] POPA, L., CHUN, B.-G., STOICA, I., CHANDRASHEKAR, J., AND TAFT, N. Macroscope: end-point approach to networked application dependency discovery. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies* (New York, NY, USA, 2009), CoNEXT '09, ACM, pp. 229–240.

[13] SNORT. http://snort.org.

[14] SOFTFLOWD. http://code.google.com/p/softflowd.

A Appendix

This appendix highlights implementation details that may be of use to network administrators interested in deploying NSDMiner on their own networks. While this appendix is written assuming that the reader will be using our open source implementation of NSDMiner, `nsdmine` [11], the higher-level concepts are applicable to any implementation.

This section is written as follows: we begin with a discussion of NSDMiner's expected input and how to configure a network to collect this input. We then discuss the parameters for NSDMiner's various algorithms with backreferences to the sections in the paper where they were first introduced. Finally, we conclude with the expected output and how to interpret this output. For any details omitted from this section, we refer the reader to NSDMiner's README file.

A.1 Preparation

In order to run NSDMiner, network traffic flows must be collected and stored. This can be accomplished in several ways. Some routers have the ability to save and export Cisco Netflows (using the switch `--cisco`), which are a standard format for representing network traffic flows.

For routers that do not export Cisco Netflows, it is also possible to use port mirroring on routers to copy and forward all traffic to a central location, and use a tool such as *softflowd* [14] to reconstruct flows from the individual packets. This may already be set up on networks running an intrusion detection system such as *snort* [13], in which case it should be possible to configure the IDS to save packets locally to be later processed by NSDMiner.

Our implementation of NSDMiner supports loading pcap (tcpdump) packets by using the `--pcap` switch. We implement NSDMiner as a Python package and represent flows as an object, meaning it is possible to create a loader that can convert network traffic into the format needed by our implementation without having to change the code for the NSDMiner algorithm itself.

The amount of data is needed to use NSDMiner usefully is a function of the size and the activity levels of the network. On our single-building network, it took 2-3 weeks of data to reach the point of diminishing returns [10].

A.2 Choice of Parameters

NSDMiner accepts the following parameters, presented as switches to the `nsdmine` invocation.

- A set of IP addresses I referring to all of the servers for which dependencies should be collected, provided as a comma-seperated filter of IP addresses.
- The confidence threshold α, between 0 and 100. Dependency candidates with a confidence value (as calculated by the algorithm in Section 3.1) less than α will be pruned from the output.
- The similarity threshold δ, between 0 and 100, used for the Inference algorithm (Section 3.2).
- The agreement threshold Δ, between 0 and 100, used for the Inference algorithm (Section 3.2).
- The support threshold σ, an integer greater than 0, used for the Clustering algorithm (Section 3.3).
- A confidence threshold α', an integer greater than 0, and less than α.
- An example of an invocation to NSDMiner setting all of these options would be in the form `nsdmine --filter=`I `--alpha=`α `--infer=`δ,Δ `--clusters=`σ,α'.

NSDMiner works in two phases, the first being dependency graph generation, followed by post-processing. Dependency graph generation analyzes the network flows as explained in Section 2.2. Generation of the dependency graph runs in $O(n^2)$ time, and because the input is generally in hundreds of millions of flows, this can take a few days to generate the entire dependency graph. Specifying exactly which servers should be monitored for dependencies substantially reduces this time.

The second phase is post-processing, where inference (`--infer`), clustering (`--clusters`), and ranking are performed on the dependency graph. Each post-processing component accepts configurable parameters that will vary in effectiveness from network to network. Inference attempts to identify the dependencies of services that are infrequently used by comparing them to similar services in the network. Inference is only useful when there are several services that are configured in the same way, such as in a data center or web-hosting provider's server farm. δ should be set high if NSDMiner is being used on a noisy network. However, if one is analyzing a private LAN, δ can be set lower. Δ should be set high if many of the services in the network are part of clusters, and low if most of the services are only hosted on single machines.

The clustering algorithm attempts to locate redundant service clusters, such as those used for back-up or load-balancing purposes. Clustering can only find service clusters that are used by more than σ services. The setting of σ is much like a peak-detection problem – it should be high enough to ensure that all coincidental traffic is ignored, but low enough to catch all of the actual clusters. α' should be set to about half of a α.

Finally, the dependencies are ranked and given confidence scores normalized from 0 to 100%. Any dependencies with a score less than α will not be present in the final output. Each of the post-processing approaches work in parallel on the original dependency graph generated in phase one, so their ordering does not matter.

A.3 Output

The final output is the list of services running on the machines in I and each of their dependencies, each marked with a confidence ranking from α to 100%. The ranking roughly estimates the probability that the specified dependency is a true positive. If clustering was used, then the clusters that were found will also be reported.

After collecting this input, the operator should then verify each of the reported dependencies manually to determine how accurately NSDMiner runs on his or her network. By verifying the output, it is possible to fine-tune the parameters to make using NSDMiner more reliable in future runs.